Automated Privacy Audits Based on Pruning of Log Data

Rafael Accorsi and Thomas Stocker
Department for Telematics
University of Freiburg, Germany
{accorsi,stocker}@iig.uni-freiburg.de

Abstract

This paper presents a novel approach to automated audits based on the pruning of log data represented as trees. Events, recorded as a sequential list of entries, are interpreted as nodes of a tree. The audit consists in removing the nodes that are compliant with the policy, so that the remaining tree consists only of the violations of the policy. Besides presenting the method, this paper demonstrates that the resultant method is more efficient than usual audit approaches by analyzing its theoretical complexity and the runtime figures obtained by a proof of concept.

1. Introduction

An audit is independent, unbiased analysis of the activity of the system. The goal of an audit is to ensure that the system acted in compliance with the governing policies, holding sources of misbehavior accountable in case of incompliance. Privacy audits constitute a special form of audit where the access and usage of data items are analyzed against the privacy policy of data providers.

The need for automated, efficient analysis of log trails is evident in a number of scenarios [3, 12]. The reasons are two-fold. First, state-of-the-art privacy policy languages encompass, together with traditional authorizations, also obligations, i.e. postconditions that must be fulfilled after the authorization. Because obligations are generally not enforceable by execution monitors at runtime [7], the adherence to obligations can only be determined a posteriori by means of audits [2]. Second, besides providing a mechanism for sound internal control [6], when used at the interface between the system and data providers, the availability of privacy audits improves the transparency of the system. Consequently, knowing that they can obtain information about the usage of released data, data providers fell more confident when interacting and releasing personal information to enterprises. Both cases anticipate an authentic audit trail recording the execution of the system. To this end, secure logging mechanisms are used [1].

Despite their raising relevance for enterprise computing, to-date privacy audits are at best semi-automated [3]. Studies show the implications of manual steps in audits, demonstrating that the examination of simple audit trails may take months to be completed [13]. Manual steps are not only time consuming, they are also sources of errors eventually undermining the whole audit process and its findings.

Contribution. This paper presents a novel approach for automated privacy audits based on pruning. Instead of operating on raw, sequentially recorded log data, log files are transformed into tree representations in a preprocessing step. These trees borrow their hierarchical structure from partial orders used to characterize the objects and subjects in state-of-the-art policy languages based on XML dialects, such as E-P3P [4] and ExPDT [8]. Taking a tree and the corresponding privacy policy, the privacy audit successively prunes the tree, removing the nodes (and their child nodes) that are compliant with the policy rules and leaving those nodes that are incompliant. The remaining tree thus comprises the policy violations found during the audit, whereas an empty tree means that no violation has been detected.

Analyzing the complexity of privacy audits based on pruning, this paper also demonstrates this approach is efficient. Specifically, although the proposed audit algorithm has a quadratic complexity with the number of entries in the audit trail, asymptotically, it exhibits an almost linear growth for sizes of audit trails found in the practice up to 1000K entries per user.

Based on this analysis, this paper also investigates the relationship between the efficiency of pruning audit and the complexity of the tree structure of audit trails, as well as the relationship between the efficiency of pruning audit and the number and kind of policy rules by which audit is parameterized. Regarding the complexity tree structure, more detailed hierarchies of subjects and objects lead to faster audits. Regarding the influence rules, the inclusion of new rules in the privacy policy leads to a constant increase of the runtime.

Related work. Approaches to audit are distinguished be-

tween *online* and *offline* modes: the former is carried out during the execution of the system in an event-driven manner; the latter is performed a posteriori, taking a truncated audit trail and analyzing it against the policies while the system still runs. While being distinct in their practical application, both approaches are theoretically equivalent with regard to the kind of policy violations they detect.

Roger et al. [11] present an approach in the online mode. Compared to the approach presented in this paper, their approach does not consider policies encompassing obligations. Moreover, audit is carried out online, thereby contrasting to the pruning-based audit, which are carried out offline. Other approaches for online audits are found in the intrusion detection community [9]. However, this setting differs considerably from the privacy setting, in particular with regard to the granularity level of policies.

Approaches to offline audit are presented in EPAL [4] and by Accorsi [2]. EPAL presents an integrated method for specifying privacy policies and audit the execution of the system with regard to these policies. However, the way audits are carried out is not defined in EPAL's documentation, rendering a detailed comparison impossible. In [2], Accorsi defines an approach for automated offline audits based on counterexamples. Similar to model checking, the rationale is to search the audit trail for counterexamples for the validity of a policy, thereby seeking to optimize the audit. This approach disregards the structure of audit trail, so that it must traverse the whole audit trails in searching for counterexample instances.

Structure. The remainder of this paper is structured as follows: §2 lays down the basis for the pruning algorithm, which is presented in §3. The efficiency of the algorithm is analyzed in §4. §5 closes the paper, providing an outlook for further research directions.

2. Assumptions and System Building Blocks

The approach proposed in this paper assumes that data providers, also referred to as "individuals", interact with systems, releasing data items (objects). The context in which this happens is called the "scenario". Subjects within the system then access these data items for a particular purpose. To protect collected data items against misuse, individuals formulate privacy policies in form of a rule set that stipulates the conditions upon which subjects are allowed to access objects and the obligations that are due in doing so. Privacy policies thus define the set of *managed data items*, i.e. data items in the scope of the rules of the privacy policy.

The activity of all subjects within the system are recorded as events in a log file. Here, it is assumed that *every* action on managed data items is recorded consistently, so that audits are parameterized by complete audit trails encompassing all the relevant events.

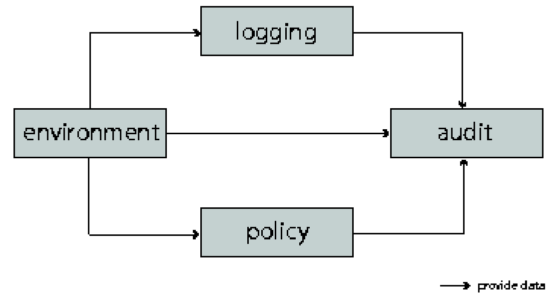

Figure 1. System components and interplay.

In order to force consistent referencing of subjects and objects, we assume that an environment is built before entries are appended to log files. This environment encompasses all data items and actors within the given scenario. The aim should be to model the concrete scenario as exact as possible, to get reliable results concerning rule violations.

The resultant audit model consists of four parts, whose interplay is depicted in Fig. 1. The *environment* component creates the domain, where not only subjects and objects are defined, but also several additional attributes subjects may exhibit, such as the purpose or the role the subject holds. This information is used while *logging* data to specify the structure of valid log entries (ensure all needed information is provided). *Privacy policies* give guidelines for accessing data items and are defined according to the environment. The level of detail of this policy depends on the expressive power of the policy language. We assume policies consisting of a set of rules, each formulating an authorization. The information provided by the environment, log files and policies are then combined in the *audit* component to determine whether the system acts in compliance with the policy rules.

2.1. The Environment Definition

The environment definition stipulates the sets of valid objects and subjects. Subjects and objects allow for some categorization and can be assigned to classes including specific characteristics and having a hierarchical correlation among each other. Thus both object-classes and subject-classes can be respectively described as a tree with a single root (*subject* or *object*). Fig. 2 shows an exemplary subject-hierarchy, including some attributes. Possible instances for this hierarchy are:

```
subject1(class: rfid)
subject2(class: thread)
subject3(class: ex_monitor)
```

The use of classes ensures that every subject or object appearing has specified attributes depending on its class. The use of classes also allows a more flexible way to define policy rules, as groups of subjects or objects can be addressed as a whole. For example, the expression

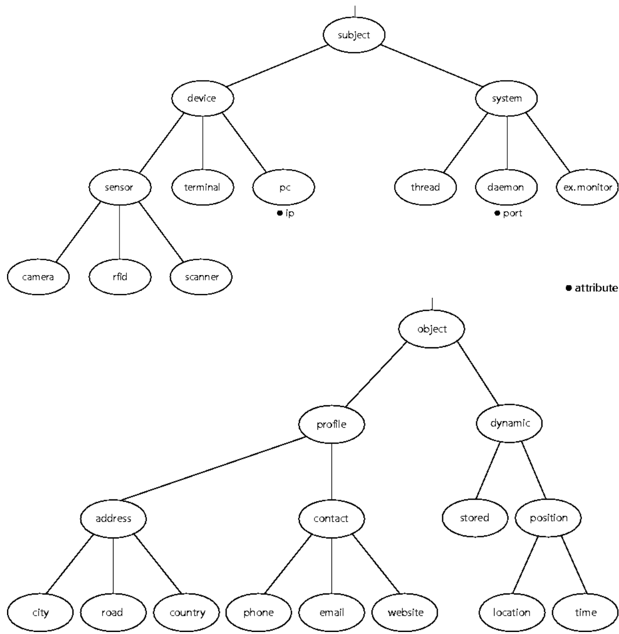

Figure 2. Example of a subject (above) and object (below) hierarchy.

```
allow(read, sensor, ...)
```

applies to all the instances of sensor, e.g. camera and RFID.

For every individual, a separate, complete object-hierarchy containing all user-specific data has to be stored. Data objects are distinguished between *static* and *dynamic* data objects. Static data items are those that do not or only marginally change along time (e.g. profile of an individual); dynamic data objects only exist during the interaction with the system (e.g. click-stream). This distinction facilitates the definition of policy rules. Fig. 2 depicts an extended object hierarchy considering static and dynamic data objects.

Given this, concrete attributes can be specified, e.g.:

```
<definitions>
   <individual>
      <attrib>pers_number</attrib>
   </individual>
</definitions>
```

The subject and object hierarchies together with the attributes of the individual are stored in a single XML-file. This file describes the sum of type-definitions made for the model as a whole. In a second step, concrete individuals and instances of subject classes can be defined. For this, every class has to offer a unique ID, which instances can refer to. This is done by assigning every node in each hierarchy a combined key, consisting of the values for his pre-order and post-order within the tree. XSLT can be used to do the key-assigning and to store the subject and object hierarchies in a separate file. To complete the definition of the scenario variables, all possible roles and purposes for later rule definitions have to be defined. One possible environment definition could be:

```
<environment>
      <individuals>
         <individual id=''1'' name=''phil''>
            <object>
               <profile>
                  <address>
                     <zip>79098</zip>
                     <country>Germany</country>
                  </address>
                  <finance>
                     <account_no>2853721</account_no>
                  </finance>
               </profile>
            </object>
         </individual>
         <individual id=''2'' name=''gus''>
            <object>
            ...
            </object>
         <individual>
      </individuals>

      <subjects>
         <subject id=''1'' h=''1'' n=''7''>
            <name>Workstation01</name>
            <attrib>
               <ip>186.44.211.34</ip>
            </attrib>
         </subject>
         <subject id=''2'' h=''3'' n=''4''>
            <name>RFID_Sensor01</name>
         </subject>
      </subjects>

      <roles>
         <role>administrator</role>
         <role>clerk</role>
      </roles>

      <purposes>
         <purpose>statistics</purpose>
         <purpose>billing</purpose>
         <purpose>data retention</purpose>
      </purposes>

</environment>
```

Parts in italic stand for free input, the rest is enforced by the XML-schema to form a valid environment definition.

2.2. Privacy Policies Definition

Privacy policies characterize the set of managed data items for each individual. To do so, privacy policies encompass rules that define which subject has which access to what object and the preconditions and postconditions that must be met to do so. Preconditions are called *provisions* and postconditions *obligations*.

The approach proposed in this paper works with any policy language based on XML, as long as it does not allow too complex obligations. For the sake of simplicity, the presentation below includes obligations defining access notification and deletion of accessed data items within a fixed time interval. Obligations concerning periodically repeated actions in future, for example, cannot be handled. Concretely,

```
<Policy>        := <Rule> | <Rule>, <Policy>
<Rule>          := (<Authorisation>; <Provision>;
                    <Obligation>)
<Authorisation> := <Subj>, <Obj>, <Right>
<Provision>     := no_prov | <Atom_Prov> |
                   <Atom_Prov> && <Provision>
<Atom_Prov>     := role == <Role> |
                   purpose == <Purpose> |
                   <Obj> == <Value>
<Obligation>    := <no_oblig | <Atom_Oblig> |
                   <Atom_Oblig> && <Obligation>
<Atom_Oblig>    := delete within <Nat> days |
                   notify within <Nat> days
<Right>         := read | write | collect |
                   execute <Cmd>
```

```
<policy>
    <rule>
        <authorisation>
            <subject h="3" n=4/>
            <object h="1" n="7"/>
            <right acton="READ"/>
        </authorisation>

        <provision type="ROLE">
            <role>3</role>
        </provision>

        <provision type="PURPOSE">
            <purpose>8</purpose>
        </provision>

        <provision type="CONSTRAINT">
            <field>IP-Address</field>
            <value>144.43.76.122</value>
        </provision>

        <obligation type="DELETE">
            <days>10</days>
        </obligation>

        <obligation type="NOTIFY">
            <days>1</days>
        </obligation>
    </Rule>
</Policy>
```

Figure 3. BNF for the policy language (above) and an exemplary policy (below).

Fig. 3 depicts the BNF of the policy language we employ as well as an example of privacy policy.

Two aspects of this policy language must be emphasized. First, while object and subjects are organized in hierarchies, roles and purposes are not, so that only the equality operator is applicable for these preconditions. Second, the policy language assumes a *default-deny* ruling. This means that every action that has not been allowed explicitly by a policy rule is automatically denied.

To represent this policy-language in XML, a schema is needed that validates a given XML file as a well-formed privacy policy, i.e. one that adhered to the BNF in Fig. 3. XML-Schema is not applicable for this policy language, for it cannot handle nondeterministic structures. This is required, as provisions differ in their structure (constraint-provisions, i.e. those defining constraints on subject attributes vs. simple provisions, i.e. those merely requiring a role or purpose). This problem can be circumvented by other schema languages, e.g. RelaxNG.

```
<history>
  <entries>
    <entry action="GRANT" time="1385345" type="READ"
           subject_id="5" suject_h="1" subject_n="7"
           object_h="5" object_n="4"
           role="2" purpose="1"/>
    <entry action="DENY" time="1385387" type="WRITE"
           subject_id="34" suject_h="2" subject_n="10"
           object_h="1" object_n="18"
           role="8" purpose="3"/>
  </entries>
</history>
```

Figure 4. Excerpt of a log file.

2.3. Logging Events

For a correct audit a complete logging of all occurred events is essential. In this context, it is assumed that every subject acts within a certain role and has one or more purposes. Every log entry has to provide information about the concrete subject, the class, role and the data object it accessed and for which purpose. Basically, there are two types of log entries: either an entry represents the execution of an action or it reports the denial of an action. This is distinguished by *GRANT* and *DENY*. Further, it is assumed that the log entries of each individual are stored in a separate file, which have to be protected against unauthorized manipulation. (See [1] for an approach to protecting log data against manipulation.) Fig. 4 shows an excerpt of a possible log file, whose entries are represented in XML.

3. Conducting Privacy Audits

This section focuses on the pruning algorithms responsible for the execution of privacy audits audits.

The pruning audit is carried out in two steps, as depicted in Fig. 5. The first step is a preprocessing one, transforming the log file into a tree structure. The second step consists of pruning the resultant tree according to the policies.

The execution of the first step considers the following parameters.

- r, the number of policy-rules
- n, the number of log-entries
- h, the height of the object-hierarchy
- m, the number of child-nodes on average per object-node within the object-hierarchy
- k, the number of object-classes (derived from h and m)

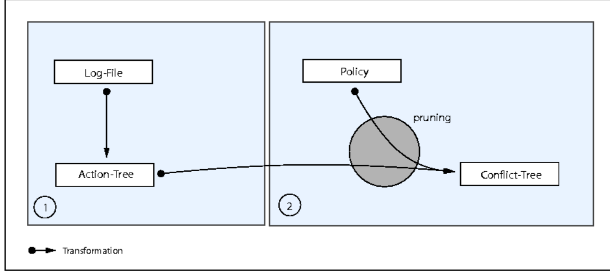

Figure 5. Structure of the audit component.

- *s*, the number of subject-instances
- *p*, the number of provisions per rule
- *c*, the number of constraint-provisions per rule
- *o*, the number of obligations per rule

The first step transforms the log file into a tree-like data structure. The resultant tree is called "action-tree". In detail a tree representing the object-hierarchy is taken to then place the log entries at the corresponding nodes, where nodes denote object classes. As every log entry keeps the information about the object-class it applies to, this can be done for all log entries. Considering the given parameters, this is done in *O(h*n)* steps. Because *h* does not depend on *n*, this is actually linear time. To this end, Alg. 1 defines a divide-and-conquer (D&C) approach.

The algorithm recursively calls the function ACTION-TREE. Within every call $O_{current}$ is the actually processed node within the object hierarchy. $E_{current}$ stands for the remaining Log-Entries. The function PRINT is used to write output. Alg. 1 then decides which log entries belong to the actual node and respectively to his child nodes. Depending on this mapping, $E_{current}$ is partitioned using the function PARTITION. MENTRIES(x) returns the corresponding partition for node x created in PARTITION. At first the algorithm creates an opening tag for $O_{current}$, then it writes out all corresponding log entries using MENTRIES$(O_{current})$. In the next step it recursively calls the function ACTION-TREE for every child node whereas the belonging partition is provided additionally. Once every node in the object hierarchy has been observed, the algorithm stops and outputs the action tree. To do so, the algorithm has to be started with the root node of the object hierarchy and the complete log file as initial inputs.

The use of pre-order and post-order values to index the nodes within the object hierarchy enables the algorithm to decide, in every step, which entries correspond to which child node or its successors while partitioning the actual entry list. Without this indexing, the algorithm would have to observe the subject hierarchy to gain this information.

Algorithm 1 Action-Tree Transformation

```
1:  function ACTION-TREE(O_current, E_current)
2:      PRINT(< O_current >);
3:      PARTITION(E_current);
4:      PRINT(MENTRIES(O_current));
5:      for all O_new in CHILD(O_current) do
6:          ACTION-TREE(O_new, MENTRIES(O_current));
7:      end for
8:      PRINT(< /O_current >);
9:  end function
10: ACTION-TREE(rootNode(ObjectHierarchy), LogFile);
11:     ▷ PARTITION distributes E_current among the actual
    node O_current and its child nodes.
12:   ▷ MENTRIES provides the corresponding partition of
    E_current generated by PARTITION.
```

```
<object h="0" n="11">
    <object h="1" n="5">
        <events>
            <entry action="DENY" time="10001" type="READ"
                   subj_id="2" subj_h="2" subj_n="2"
                   object_h="1" object_n="4"
                   role="1" purpose="10"/>
            <entry action="GRANT" time="10006" type="READ"
                   subj_id="2" subj_h="1" subj_n="2"
                   object_h="1" object_n="4"
                   role="2" purpose="4"/>
        </events>
        <object h="2" n="1">
            <events>
                ...
            </events>
            ...
        </object>
        ...
    </object>
    ...
</object>
```

Figure 6. Example of an action-tree.

So, even if no entries match the currently processed node, the input set can be subdivided and propagated to the child nodes. This transformation is thus performed in $O(h * n)$. Using a D&C approach eases the adoption of the algorithm to concrete tree-traversal languages, such as XSLT. The use of XSLT has, however, a downside: due to several technical restrictions with regard to the treatment of variables, XSLT downgrades the the runtime to *O(h*m*n)*. This is still linear in time, but it worsens the runtime by a factor of *m*. Fig. 6 depicts an excerpt of an action tree generated by Alg. 1.

The second step of the audit is the pruning. In this step, all the allowed actions are removed (pruned) from the tree, leaving a rest which exactly corresponds to the sum of all rule violations. To this end, we propose an algorithm which sequentially processes all rules within the policy and searches for the node in the action tree corresponding to the referenced object class in the rule. For a balanced tree

Algorithm 2 Pruning

```
1:  for all rules R in policy do
2:      Obj_match ← FINDOBJ(r.object);
3:      for all entries E in Obj_match do
4:          if E matches with R then
5:              violation ← CHECKTRIVIALCOND;
6:              for all constraint-provisions C do
7:                  check ← false;
8:                  S ← SEARCHSUBJECT;
9:                  check ← CHECKCONSTRAINT(S, C);
10:                 if not check then
11:                     violation ← true;
12:                 end if
13:             end for
14:             for all obligations O do
15:                 check ← false;
16:                 for all successors of E do
17:                     check ← ISOBLIGATION;
18:                     if ckeck then
19:                         break;
20:                     end if
21:                 end for
22:                 if not check then
23:                     violation ← true;
24:                 end if
25:             end for
26:             if not violation then
27:                 PRUNE(E);
28:             else
29:                 MARKASVIOLATION(E);
30:             end if
31:         end if
32:     end for
33: end for
34:     ▷ trivial Conditions are role- and purpose-provisions
```

representing an object hierarchy, the search takes at most h steps. When this node is found, the actions it comprises have to be checked against the applicable policy rule. As a consequence of the default-deny assumption, entries denoting actions not specified in any policy rule are categorized as violations. The pruning method is defined in Alg. 2.

As mentioned above, the algorithm processes a given policy rule by rule. It does so by extracting the index of the referenced object and searching it within the action tree. This is done by the function FINDOBJ$(index)$. After this, the algorithm checks all log entries contained in the object node Obj_{match} if they violate some rule conditions.

Assuming that the actions are equally distributed among the object classes, there are $\frac{n}{k}$ actions within each class. Hence, $\frac{n*(m-1)}{m^{h+1}-1}$ actions have to be processed. (Note that k is derived from $\frac{1-m^{h+1}}{1-m}$) First, the algorithm checks, if the log entry found matches with the rule by checking the values for the type of access and the referenced subject class. Now, every matching action has to be tested against all criteria specified in the current rule. If an action does not match, it is not processed further.

In Alg. 2, every provision, except constraint-provisions, can be checked immediately, as all the fields (attributes) required for this check are present in the action. The check of constraint-provisions and obligations is more elaborated. A constraint provision asks for certain attribute values of the accessing subject. To check them, this subject must first be searched in the environment data where all the subject instances are listed. This is done by the function SEARCHSUBJECT. Assuming that there are *s* subject instances within the environment ordered by their *id*, it takes *log(s)* steps to find a given subject. CHECKCONSTRAINT then checks whether the subject exhibits the correct value.

Obligations are harder to check, as they ask for certain log entries to appear in the future corresponding to a performed access. If obligations are present, these actions can be found amongst the successor actions within the actual object class. Obligations are limited to the types NOTIFY and DELETE, which are attributes of log entries representing the notification about a fulfilled obligation. For each subsequent log entry, a simple attribute value comparison therefore suffices, which is carried out by the function ISOBLIGATION. As stated above, there are $\frac{n*(m-1)}{m^{h+1}-1}$ actions per object class on average. In the worst case, no obligation holds. This means that, for every action, all successor actions have to be checked. This results $\frac{n*(m-1)}{2*(m^{h+1}-1)}$ steps for each obligation on average.

If no violations could be detected along all the checks, the observed log entry can be pruned, otherwise it is marked as a violation. By marking a log entry as a rule violation, further information about the reason(s) can be added. The function MARKASVIOLATION is used for this purpose. Summing up all the processing steps results to an overall step count S such that $S(n) = r*h*\frac{n*(m-1)}{(m^{h+1}-1)}*((p-c)+c*log(s)+o*\frac{n*(m-1)}{2*(m^{h+1}-1)})$.

To obtain a more concrete evidence on the behavior of the algorithm and overall method, the following scenario is assumed (this scenario is considered to be realistic in industrial audit settings [10]).

1. Object-hierarchies are not arbitrary in their size. Typical hierarchies do not exceed a height of 6 and an average child-node-count of 4, so that $h = 6$ and $m = 4$.

2. Nodes within the object hierarchy which are not leaves typically serve just for classification. That is, they do not contain any actions in the action-tree. Thus, the search for a certain object class takes exactly h steps

and the amount of actions within any object class of interest exceeds. It can be shown that this increase is characterized by $\frac{k}{k-1}$. So, assuming a typical object hierarchy in the sense of Item 1., the amount of actions increases by a factor of 1.34.

3. Traditionally, a policy rule does not contain more than 2 constraints and 1 obligation (see [5]). The following assumes one of these 2 provisions is a constraint provision, i.e. $p = 2$ and $o = c = 1$.

4. A policy containing 10 rules is assumed

5. Log files containing up to 1000K entries are assumed.

Based on this setting, the step formula yields $S(n) = 0,000001*n^2+n*(\ln(s)*0,015851+0,010987)$. The value of $\ln(s)$ is estimated by 10 (e^{10} instances). It follows that $S(n) = 0,000001 * n^2 + 0,169496 * n$. Because the action tree has to be created before pruning is possible, the computational effort needed to generate this tree must be considered. This results in an overall complexity characterized by: $S(n) = 0,000001 * n^2 + 6,169496 * n$.

Since we are dealing with the worst case, this is still an upper bound. According to this formula, the complexity is in class $O(n^2)$. However, the actual steps are much nearer to a linear function than to a parable for any practical case. This is indicated by the gradient S' of S, which is $S'(n) = 0,000002 * n + 6,169496$.

We compare our results to a bisecting line by observing the ratio of both functions, which is characterized by:
$R(n) = 0,000001 * n + 6,169496$
In particular, it can be seen that the ratio starts with a value of 6,16946 and increases very slowly. In the case of maximal 1000K log entries, this value only goes up by 1,00592. So, we can expect that for practical cases, the steps for carrying out an audit is almost linear. Another way to show this is to compare the step function to a concrete linear function. In this example we use $L(n) = 10 * n$ for this purpose. Fig. 7 illustrates this, showing both functions and the point where they meet.

This is a surprising result, demonstrating the efficiency of the pruning. Intuitively, one would not expect the step function scales like in an almost linear manner. However, it should be noted that this algorithm does not take into account the intricacies of the implementation.

This can be interesting to determine for example false positives. Because the audit-result is provided in form of XML, it is straightforward to transform it in a more readable representation using, e.g., XSLT. If Alg. 2 is extended by the collection of violation details, it should be possible to give a detailed overview of all violations, including a brief justification of what went wrong and why.

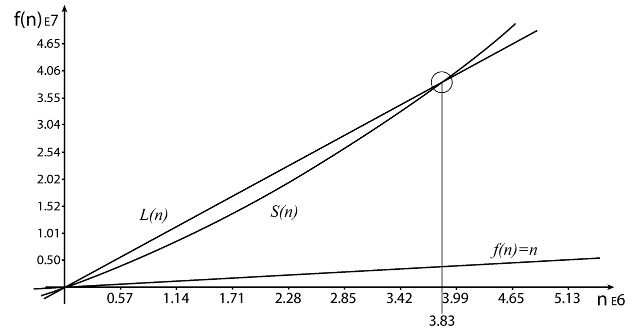

Figure 7. Comparison of S with the linear function L and a bisecting line.

4. Evaluation

Although the results stated above already indicate the efficiency of the pruning approach to audit, it would be interesting to compare its step count to another approach to audit, thereby measure its effectiveness.

We consider the standard algorithm for audit, which sequentially traverses a log file and checks, for every entry, whether any rule is violated. By checking a rule, there are possibly constraint-provisions and/or obligations which cannot be tested trivially. For constraint-provisions, the standard algorithm takes the same number of steps as the pruning algorithm, i.e. $log(s)$. An obligation requires the investigation of all subsequent entries within the log file. This takes $\frac{n}{2}$ steps on average. Therefore we get a step function for the standard audit algorithm such that:
$S(n) = n * r * (log(s) + \frac{n}{2})$,
Using the assumptions for industrial cases, this step function is simplified to $S(n) = 5*n^2+144,27*n$. Given that, we now determine the percentage of steps (*P*) the pruning approach needs compared to the standard approach, which results in
$P(n) = (\frac{1,234}{n+28,85} + 0,0000002) * 100$
Such a *P* shows that already for a log file with 95 entries the pruning approach only needs 1% of the steps the standard approach does.

Parameter modification. For realistic applications, it is important to analyze the effect of parameter modifications to the overall step count. Generally, we are interested in the gradient of the first derivative of the step function $S(n)$, for it shows how fast the step function grows. For this second derivative, only the factor of n^2 matters. Hence, we separate this factor by deforming. Additionally we have to add a factor of 2 for the first derivative. We obtain:
$S''(n) = \frac{2*h*r*(m-1)^2}{(m^{h+1}-1)*(2*m^{h+1}-1)}$.

The parameter *r*, which stands for the number of rules,

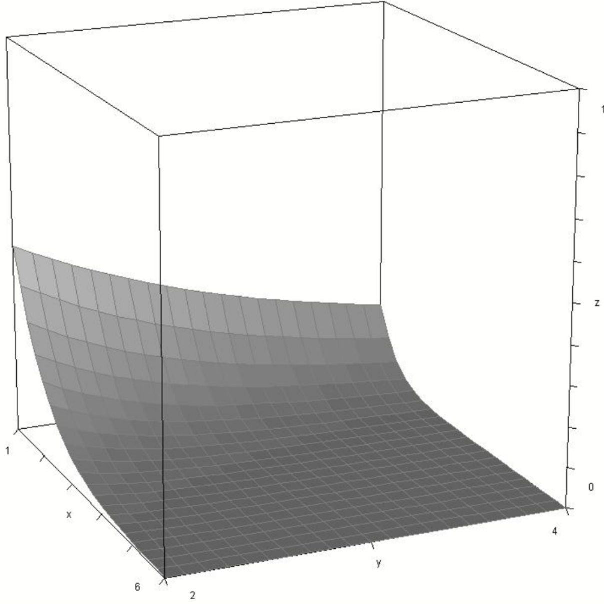

Figure 8. The combined factor of h and m, with $x = h, y = m$ and z serving as factor

can be separated as a single factor. Hence, an increase of *r* causes the difference between the new value for $S''(n)$ and the old one increases the same way, i.e. increasing the number of rules does not lead to an increase of the computational effort needed to carry out the audit using the pruning approach, which is a relevant result.

For the parameters *m* and *h*, which stand for the number of object classes and the height of the tree, a bit more of work has to be done. Due to restricted space, the mathematical details are omitted. However, it can be shown that the rise of either *h* or *m* lead to significantly lower step counts. Fig. 8 shows how the term $\frac{h*(m-1)^2}{(m^{h+1}-1)*(2*m^{h+1}-1)}$ behaves.

That is, the more complex the hierarchies encoded by the tree structures are, the more efficient is the audit. Although we expected some outcome by using hierarchies, the overall results obtained in this approach are far beyond our expectations. In particular, the gain of efficiency compared to the standard audit approach is significant. To take advantage of this, it is recommended to build a detailed object hierarchy at the definition of an environment.

5. Conclusion and Outlook

This paper presents a novel approach to automated audits based on the pruning of log data organized as trees in form of XML documents. The audit consists in removing the nodes that are compliant with the policy, so that the remaining tree encompasses the violations of the policy. Besides presenting the method, this paper demonstrates that it is more efficient than usual audit approaches.

Nevertheless, this is just the first step in investigating efficient audit algorithms and several interesting issues remain to be examined. These issues are both of theoretical and practical nature. On the theoretical side, we still need to demonstrate the correctness of the pruning algorithm. On the practical side, it would be worthwhile to carry out case studies with real datasets, thereby showing the adequacy of the pruning method in practical systems. Specifically, it has to be studied how the proposed algorithm can be implemented without a significant complexity worsening. Although there are languages specialized for operating on trees, there are some difficulties to overcome in some cases, for example the lack of flexible variable handling in XSLT (once a value has been assigned to a variable, it cannot be changed anymore). We implemented the pruning algorithm using XSLT, but this led to substantial worse results. In this case, the choice for other languages, such as XQuery, may be more convenient. This requires further analysis, though.

References

[1] R. Accorsi. On the relationship of privacy and secure remote logging in dynamic systems. In S. Fischer-Hübner et al. (eds.), *Security and Privacy in Dynamic Environments*, vol. 201 of *IFIF*, pp. 329–339. Springer, 2006.

[2] R. Accorsi. Automated privacy audits to complement the notion of control for identity management. In E. de Leeuw (eds.), *Policies and Research in Identity Management*, vol. 261 of *IFIP*, pp. 39–48. Springer, 2008.

[3] A. Antón, E. Bertino, N. Li, and T. Yu. A roadmap for comprehensive online privacy policy management. *CACM*, 50(7):109–116, 2007.

[4] P. Ashley, S. Hada, G. Karjoth, and M. Schunter. E-P3P privacy policies and privacy authorization. In *WPES*, pp. 103–109. ACM, 2002.

[5] T. Breaux and A. Antón. Analyzing regulatory rules for privacy and security requirements. *IEEE TSE*, 34(1):5–20, 2008.

[6] A. Carlin and F. Gallegos. IT audit: A critical business process. *IEEE Computer*, 40(7):87–89, 2007.

[7] K. Irwin, T. Yu, and W. Winsborough. On the modelling and analysis of obligations. In *CCS*, pp. 134–143. ACM, 2006.

[8] M. Kähmer and M. Gilliot. Extended privacy definition tool. In *MKWI*, LNI. Springer, 2008.

[9] T. Lunt. Automated audit trail analysis and intrusion detection: A survey. In *11th CSC*, 1988.

[10] P. Maier. *Audit and Trace Log Management*. Auerbach, 2006.

[11] M. Roger and J. Goubault-Larrecq. Log auditing through model-checking. In *CSFW*, pp. 220–235. IEEE, 2001.

[12] S. Sackmann, M. Kähmer, M. Gilliot, and L. Lowis. A classification model for automating compliance. In *CEC08*, IEEE, 2008.

[13] P. Waterfield and J. Casey. The governance of compliance: Putting policies into practice. Yankee Report, April 2005.