Invalidating Policies using Structural Information

Florian Kammüller[1]* and Christian W. Probst[2]
[1] *Middlesex University, UK*
f.kammueller@mdx.ac.uk
[2] *Technical University of Denmark, Denmark*
cwpr@dtu.dk

Abstract

Insider threats are a major threat to many organisations. Even worse, insider attacks are usually hard to detect, especially if an attack is based on actions that the attacker has the right to perform. In this paper we present a step towards detecting the risk for this kind of attacks by invalidating policies using structural information of the organisational model. Based on this structural information and a description of the organisation's policies, our approach invalidates the policies and identifies exemplary sequences of actions that lead to a violation of the policy in question. Based on these examples, the organisation can identify real attack vectors that might result in an insider attack. This information can be used to refine access control systems or policies. We provide case studies showing how mechanical verification tools, i.e. modelchecking with MCMAS and interactive theorem proving in Isabelle/HOL, can be applied to support the invalidation and thereby the identification of the attack vectors.

Keywords: organisational structure, policies, formal methods

1 Introduction

Insider threats are a major threat to many organisations. Insiders have special privileges and knowledge, which not only enable them to perform many actions unobserved, but also make them interesting for attackers outside an organisation. The literature shows various examples [1–5].

Even worse, insider attacks are usually hard to detect, especially if an attack is based on actions that the insider has the right to perform. In this case, finding the legal but maliciously intended action is as hard as finding the proverbial needle in a haystack.

A natural reaction to this kind of threat would be to try to ensure that after an attack one has sufficient proof to identify the attacker, and to remedy the actions of the attack. The problem however is to decide what to log. Too much information makes it impossible to identify important pieces (the above mentioned haystack), but too little information makes it unlikely that there is a needle to find.

In this paper we present a step towards detecting possible insider attacks by invalidating policies. Invalidating policies is not a new idea. What is new in our approach is that we take structural information into account for guiding the invalidation. This structural information will usually be in form of a model of the part of the organisation being analysed, *e.g.*, a system model representing the organisation's infrastructure, or a representation of a workflow. Based on a model of the organisation and a description of the organisation's policies, our approach identifies exemplary sequences of actions that lead to a violation of the policy in question. We further propose to use well-established mechanical verification tools for modeling systems and workflows of organisations, use the tool support to invalidate them by negating policy properties or checking for exemplary paths leading to invalid states. Using the exemplary sequences of

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, volume: 5, number: 2, pp. 59-79
*Corresponding author: Department of Computer Science, Middlesex University, Town Hall, The Borroughs NW4 4BT London, UK, Tel: +442084114930

actions found, the organisation can identify real attack vectors that might result in an insider attack. This information can be used to refine access control systems, policies, logging, or combinations hereof.

Both approaches presented here are based on some formalization of policies. The formalizations and our results show that logical policy formalization can be a good starting point to invalidate policies, and may also be taken as a starting point for a systematic design of, *e.g.*, access control or policy systems.

The rest of this article is structured as follows. After discussing policies and their roles for insider threats in the next section, we present two approaches to invalidating policies based on structural information: the first uses system models (Section 3), the second uses workflows (Section 4). We demonstrate next how to use the model checker MCMAS [6] to invalidate security policies by building the system model and checking whether an invalid state can be reached (Section 3.3). Both example scenarios for insider attacks (the Janitor and the road-apple attack) can be identified. For workflows, however, we propose the use of the interactive proof assistant Isabelle/HOL [7] to logically invalidate a policy (Section 4.2 and Section 4.3) and to apply refinement for an even deeper invalidation strategy (Section 4.4). We use a real case study of the District of Carolina Tax Fraud (Section 4.1) [8, 9]. After a discussion of related work in Section 5, we conclude in Section 6.

2 Policies

Policies describe admissible or inadmissible behaviour in organisations. As such they also are (if complied to) an obvious means for regulating insider threats; at the same time research indicates that there is an upper limit to the number of policies that employees will comply with [10].

Since insiders will usually have good knowledge of policies and may use this knowledge combined with their respective access rights granted by those policies to circumvent regulations, the policies are a good starting point to explore insider attack possibilities. When we thus focus on the policies, the models of the actual workflows become a parameter whose level of detail we use to explore to what extent a given policy can be violated. We consider policies as restrictions on dynamic state based system models and additionally as logical constraints describing workflows of organisations. Full state exploration as well as simple propositional logic evaluation serve to invalidate these policies and thus exhibit attacks.

The two sections Section 3 and Section 4 currently each use their own policy language. We are in the process of unifying these two branches, and to investigate their compatibility with policy languages such as XACML.

3 System-model based Policy Invalidation

In this section we discuss how to use structural information from system models [11, 12] to invalidate policies. The system models we consider are based on ExASyM [11].

As argued above, structural information can be instrumental in strengthening the invalidation of policies. The system models we consider describe especially access control specifications of the organisation in question, and these specifications can be used in the invalidation process to identify insiders that might have performed a certain action. It should be noted that for many actions we can not clearly identify a certain actor as source; instead, the source of an action will most often only be describable by a set of capabilities, *e.g.*, a certain key or access rights. We plan to explore this in future work.

3.1 Organisation Infrastructure Example

In Figure 1 we show an example for the infrastructure of an organisation based on [11]. The infrastructure consists of two layers, the building and the computer network, some nodes of which are collocated, for

example the computers and the offices. Most rooms in the building are controlled by some form of access control, the specification of which, together with our graph-based abstraction of the model is shown in the right hand side of Figure 1. For a more detailed discussion of the underlying model see [11].

Assume a policy that prohibits data of a certain type, *e.g.*, sales data, to leave the organisation. In the example this would mean that the data is not allowed to reach either of the nodes `WWW` or `OUTSIDE`.

3.2 Invalidating Policies based on System Models

The invalidation of policies based on the system model proceeds by negating the policy, and then trying to establish a series of actions that would result in a system state that fulfills the negated policy. At the same time this system state violates the original policy, thereby giving a counterexample for the policy. As discussed above, the obtained counterexamples can be used for refining the system model and its access control specification.

To practically realize this state exploration, we can use existing methods like model checking. We use a model checker specialized for multi agent systems, MCMAS [6]. We specify the goal as the negated policy, and then let the model checker find its way to a state that violates the policy. The modeling and analysis with MCMAS is described in the following section but to illustrate the procedure of invalidation a bit more in detail, let us start by negating the policy: data should go to `WWW` or `OUTSIDE`. Let us concentrate on the second case `OUTSIDE` first. We assume as data a secret file of user U. The state exploration reveals that there are several sequences of ExASyM actions that lead to this secret file being moved to `OUTSIDE`. If the user prints the secret file, it may rest on the printer and the janitor may pick it up because he has access rights to the server/printer room. Since the printout can go to the waste bin there is no secrecy restriction any more on this printout of the secret file and the janitor can transport it to `OUTSIDE`.

If the goal state is `WWW`, similar reasoning finds out that a process at `PC1` must have sent the file, where only the user can have started the (malicious) process. Subsequently the analysis finds out that the process in question was brought in by the user from `OUTSIDE` or was received at `PC1`. This attack is especially interesting because it resembles a collection of attacks. The case of the malicious process

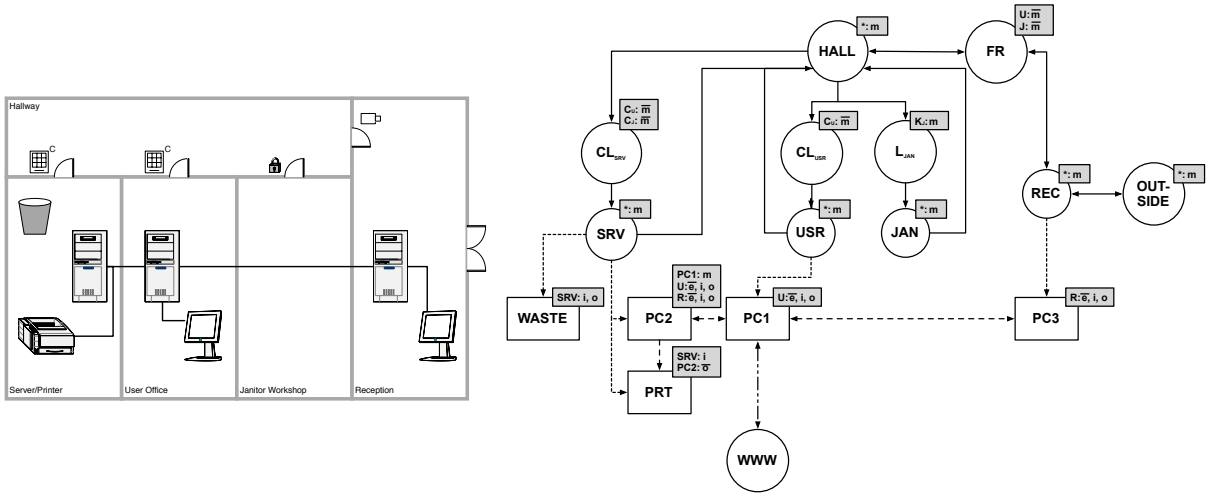

Figure 1: A simple example system and its representation as a graph in ExASyM [11], including access control annotations. The node `OUTSIDE` represents the physical world outside the organisation, and the node `WWW` represents the Internet as reachable by, *e.g.*, the mail server, here `PC1`.

being brought in from the outside represents the road-apple attack [13], the case of the process being received at PC1 represents the case of the user receiving some malware by email or picking it up at a web page.

3.3 System Model Policy Analysis with MCMAS

Usually model checkers are used to automatically prove that given models fulfill specified properties by a complete state exploration of the model. However, one very useful characteristics of model checking is that it produces counterexamples in case the examined property does not hold. These counterexamples are sequences of state changes representing a path leading from an initial state into a state violating the specified property. Equally such paths may be produced for existentially quantified temporal logical formulas, e.g., "there exists a path such that eventually p holds" – formally $\mathsf{EF}\ p$. This quality we exploit here for invalidating policies. By negating the desired policy we can then use the complete state exploration of a model checker to produce a path showing the violation state if such a path is possible. In this section, we explain how the epistemic logic model checker MCMAS [6] can be used to this end.

Epistemic Logics and MCMAS

Epistemic logics are *belief logics* which are naturally suited for the verification of security protocols, as has been exploited early on by the BAN logic [14]. Cohen and Dams show [15] that epistemic logics can naturally be employed to reason in a complete way about notion of indistinguishability central to security analysis. Observational equivalence, agent semantics based on history identity, and information flow theory can be axiomatised [16]. As a consequence, BAN and its successor SVO can be embedded in epistemic logics [15]. The epistemic logic model checker MCMAS, `http://www-lai.doc.ic.ac.uk/mcmas/`, contains an expressive subset of CTL* augmented with epistemic logic. Thereby, temporal properties may be specified together with properties containing "knows" statements. Equal to other modal logics, the world is modelled as a directed graph over states. Epistemic logic additionally has a set of *agents* whose knowledge is defined by the propositions of each world. The modal operator, K (for "knows"), quantifies propositions over agents, for example $M, w \vdash K(A, p)$ means in world $w \in M$ agent A knows p. The formula is interpreted, similar to the universal A operator in CTL*, as: p holds in w and all possible worlds w' that may be reached from w.

Janitor in MCMAS

In a preliminary study [17] we have shown how MCMAS can be applied *ad hoc* to the Janitor example for a simplified scenario (`OUTSIDE` was not part of this scenario). Interestingly, we do not use any of the knowledge operators to express properties for our attack examples but we heavily rely on the agent based modeling language of MCMAS. It is very nicely suited for expressing system models for insider attack scenarios as they come from ExASyM models. Note, that we do not yet prescribe a systematic translation from ExASyM to MCMAS. However, the following system models indicate a recipe for representing ExASyM system models in epistemic logics. The full MCMAS code for the Janitor example is contained in Appendix A.1 including a sample output of the model checker illustrating the above mentioned generation of a path leading to an invalidating system state. We explain now the guiding principle of the MCMAS representation making occasional reference to the concrete specification code contained in the Appendix. The Janitor specification uses two agents, `Agent User` and `Agent Janitor`. The system graph representing the physical scenario of the system model is modeled using agents' state variable `currentposition` ranging over the set of values {`outside, hall, pc, server`} for agent `User` and {`outside, hall, server, janitor`} for agent `Janitor`. The state change, i.e., the moving and security related actions of the actors in the physical model is specified by the agents' `Actions`.

The actions for the User are print, movein, move, move1, move2, moveout representing for the most part the moves between rooms (move1 is the move to the server room and move2 to the PC room) and one action print with the intuitive meaning that the user prints the secret file. The actions are conditional on the current state which is expressed in MCMAS in the Protocol section of an agent. For example, the following clause from the User's protocol section defines that action movein can only occur when the current state represents the outside location.

```
currentposition = outside : { movein };
```

The state changes depend on previous states and current actions of the agent. For example, the user can only print the secret file, if he is in the PC room. This is represented by the following clause in the Evolution section of Agent User.

```
currentposition = pc if (currentposition = pc and Action = print);
```

The specification of agent Janitor has similar elements for recording this agent's position and specifying the actions and the state changes. However, the Janitor has an additional action pickfromwaste and a boolean variable has_secretfile that acts as a data latch recording in the state of the Janitor that he has succeeded and picked the printed secret file from the waste bin. This can happen if the Janitor happens to be in the printer room when the User prints out the secret file. Formally, this crucial part of the specification is encoded in the following clause in Agent Janitor.

```
has_secretfile = true if (currentposition = server and User.Action = print and
                          Action = pickfromwaste);
```

The properties that need to be checked are defined in the Evaluation section of the MCMAS specification. For the Janitor, we have the following two properties that express first that the Janitor succeeds in picking up the printout of the secret file and second that he succeeds in walking outside with it.

```
janitor_succeeds if Janitor.has_secretfile = true;
file_outside if (Janitor.has_secretfile = true and Janitor.currentposition = outside);
```

Next, we need to define the initial states for the agent's and their state variables. The final and central model checking process is encoded in the Formulae section of the MCMAS code file. The following three properties are contained there of which the first two check as true while the last one checks as false.

```
EF(janitor_succeeds);
EF(file_outside);
AG(!janitor_succeeds);
```

The first property expresses that the Janitor is able to go to the print room and wait there for the printout of the secret file to be printed by the user to pick it up "from the waste bin" (which is the reason why he has access at all to the print room). The second property subsumes the first success and extends it to the Janitor being able to consecutively walk out. Both properties are quantified by the existential path quantifier EF meaning that paths exist on which eventually the respective property becomes true. Equivalent to such existential properties representing the possible invalidation of a desired security property is the explicitly negated and universally quantified invalidation (with AG, i.e., "all globally"). Logically, the last formula is the inverse of the first formula. This becomes practically visible when we look at the output. The first formula checks true and the last checks false but the system state paths are the same for the two (see Appendix A.1.4).

To prohibit the attack, the system model can be refined, for example, by introducing a refined lock mechanism at the door to the server room that would activate time delays on access for the janitor and user alerts as pickup reminders when secret files are printed.

3.4 Road Apple Attack in MCMAS

The road apple is a usb stick that contains a malicious process. Alternatively, this malicious process can be on a web page or in an email. The system model in MCMAS is contained in Appendix A.2. We model the user in a very similar fashion as in the Janitor example but instead of a `print` action the `Agent User` has a `download` and a `pickup` action representing the download on the malicious process and the picking up of the usb stick by the user, respectively. We could have integrated the Janitor and the road apple attack into one MCMAS model but for simplicity and efficiency we use two separate specifications; the agent Janitor can be omitted in the road apple attack.

The malicious process is modelled as an additional agent `Malex`. This main design decision enables the independent treatment of the process as an entity that acts like an attacker. It sleeps on the key and if the agent `User` picks it up it stays with him.

```
pickedup: boolean;
active: boolean;
```

Eventually, the malicious process may become active which represents the success of the attack. The actions of Agent `Malex` reflect the actions of the user because once it has been picked up it accompanies the User.

```
Actions = {send_secret_data, move, move1, move2, movein, moveout};
```

The additional event `send_secret_data` represents the moment when the malicious process becomes active and sends out the captured data. The protocol for `Malex` consequently specifies that sending the secret data can only happen if it has managed to be located on the PC represented in the state by the current position `pc`. This is encoded in the following clause of the `Protocol` section of `Agent Malex`.

```
currentposition = pc1: send_secret_data;
```

The `Evolution` section specifies that the malicious process can accompany the User if the usb has been picked up. This encoding uses the latch variable `pickedup` and observes the User's actions to act accordingly. For example, the following clause specifies that the malicious process follows the User into the PC room if it has been picked up.

```
currentposition = pc1 if (currentposition = hall and User.Action = move1
                          and pickedup = true);
```

Being picked up is only possible if the User moves outside as is encoded by the following `Evolution` clause.

```
pickedup = true if (currentposition = outside and User.Action = pick);
```

The agent `Malex` goes into action if the current position is in the PC room and the User downloads it. This transition is captured by changing the state to `active` and the `Action send_secret_data`.

```
active = true if (currentposition = pc1 and User.Action = download
                  and Action = send_secret_data);
```

We express the success of this attack by a property `data_loss`.

```
data_loss if (Malex.active = true);
```

Having initialized the state variables of the agents, the MCMAS verification process checks that data loss may be possible in the following `Formulae` clause.

```
EF(data_loss);
```

The attack is provided by the output of the verification procedure describing the state evolution that leads to it.

The above specification describes the scenario of the User picking up the key and downloading the malicious code. However, the second attack possibility – that of the User downloading the code from the web or from an email – is not part of the above MCMAS system model. However, by a slight adaptation we can introduce this second attack possibility. We add the User's actions `download_from_web` and `download_from_email` and allow `active` to become true and secret data to be sent alternatively on those conditions. We replace the former `Evolution` clause by the following.

```
active = true if (((currentposition = pc1 and User.Action = download)
                    or User.Action = download_from_web
                    or User.Action = download_from_email)
                  and Action = send_secret_data)
```

The property `data_loss` remains the same and also checks true as before.

For invalidating policies for system models the agent based representation of MCMAS is very suitable. However, it may be observed that the data related modeling, e.g., secret files, downloading, and sending of data, is somewhat simplistic. Therefore, we propose an alternative method for the modeling and analysis of more data-centric or workflow-based insider attacks as follows.

4 Workflow-based Invalidation

In this section we apply a similar idea as in the previous section to the problem of invalidating policies that address workflows. As before, we have a mechanism for describing policies, and we use the additional structural information from the workflow to guide the invalidation of the policies.

4.1 Example: DC Insider Attack

Some years ago, the tax department of the District of Columbia (DC) was a target of an insider attack launched by one of their employees with the help of colleagues to undergo the policy that was in place to avoid check refund frauds [8, 9]. This employee was known as a skillful and reliable person working with IT, and used her position and knowledge of the real estate tax refund policies to undergo the security perimeter and cash in bogus check refunds.

She was also involved in setting up the scene so that their frauds would remain undetected: since costs for a new, integrated Tax System had already used up the planned resources, the insider contributed to the decision to leave out her department that handled real estate tax refunds. As a result, she could cash in bogus real estate tax refunds for fictitious parties, often collaborating with colleagues or their partners. There were a few simple policy restrictions in place that the attackers exploited to remain undetected:

- Checks below a $40000 threshold did not require a supervisor's approval, and
- there was no test in place to verify whether a check had already been cashed, so checks could be cashed-in more than once.

4.2 Invalidating Policies based on Workflows

We illustrate now that a simple logical formalization does suffice to make these fairly intuitive attack possibilities easily detectable. To formally describe the policy of the case study that has been tampered with, we introduce a datatype for checks entailing the addressed department, the recipient of the money, and the cash sum.

$$\text{check} \quad : \quad \text{department} \times \text{recipient} \times \text{sum}$$

The parts of the policies are as follows:

- *supervise* : *check* $\rightarrow$ *bool* is an abstract predicate denoting that a check refund must get the approval of a supervisor,
- *own-dept-exempt* $x \equiv x.\textit{department} = \textit{real-estate}$ is a concrete predicate defining which checks are exempt from checking because they are addressed to the real estate tax department,
- *threshold-exempt* $x \equiv x.\textit{sum} < 40000$ is a concrete predicate encoding that checks with a sum below 40000 need not be checked.

Finally, the current policy can be expressed simply as the following combination of these three predicates mandatory for all cashed checks of the DC tax department.

$$\text{policy } x \quad \equiv \quad \text{own-dpt-exempt } x \vee \text{threshold-exempt } x \ \vee \ \text{supervise } x$$

To express the global policy we assume a concrete set of "cash-ins" and then quantify the policy over all checks in the set. The operator $\mathbb{P}(S)$ denotes the powerset of a set S.

$$\begin{array}{lcl} \text{cash-ins} & : & \mathbb{P}(\text{check}) \\ \text{Policy} & \equiv & \forall x \in \text{cash-ins. policy } x \end{array}$$

4.3 DC Insider Attack Example Analysed

The logical expression of policies as discussed in Section 2 provides a tool to invalidate them. We have to analyse the ways how a fake check can pass the policy; to do so, we simply assume a fake check:

$$\textit{fake} \quad : \quad \text{check}$$

Then the following three simple properties express three ways of invalidating the policy.

$$\begin{array}{lll} 1. & \text{own-dept-exempt } \textit{fake} & \Rightarrow \quad \text{policy } \textit{fake} \\ 2. & \text{threshold-exempt } \textit{fake} & \Rightarrow \quad \text{policy } \textit{fake} \\ 3. & \text{supervise } \textit{fake} & \Rightarrow \quad \text{policy } \textit{fake} \end{array}$$

The first two invalidation properties correspond to the actual attacks that happened in the DC case. The third one has *not* taken place but represents another loophole for an insider attack: by conspiring with an insider that may act as supervisor, this additional invalidation property exhibits an additional insider threat with respect to this policy.

The analysis of the invalidation of the policy is somewhat ad hoc. It has been shown that policies can be formalized in First Order Logic (FOL) [18]. However, using Higher Order Logic (HOL), we can schematize and reason on a higher level about invalidation of policies thus rendering the above ad hoc logical analysis as a generic tool.

Although Higher Order Logic is generally undecidable, cutting edge tools like Isabelle [7] or Coq [19,20] provide sufficient automation to support applications like this example to a degree of automation. The above formalization can be one to one translated into Isabelle/HOL and the invalidation properties can be derived as very simple lemmas (see Appendix A.3).

4.4 Detecting Multiple Cash-ins

The policy and analysis presented above lack one possible attack, a flaw that has been exploited in the real case. Since check cash-ins exempt from supervisor control were not scrutinized properly, checks could be cashed in more than once. This flaw of the actual workflow persists as a weakness also in our logical specification. Since cash-ins of checks are represented as *sets* of checks, their cash-in is not audited as in real life in a sequence of actions (for example by adding time stamps to cash-ins). Therefore, a double cash-in cannot be noticed in the abstract specification.

In order to invalidate the policy, we refine it to a more concrete specification closer to the physical reality of workflows in a tax department. We use *sequences* of check cash-ins as a representation of the cash-flow of the tax department.

Double cash-ins can now be easily detected by auditing the sequence of cash-ins and while checking for double occurrences of the same check. The function count $x \in s$ is used to return the number of occurrences of x in sequence s.

$$\begin{array}{lcl} \text{cash-ins} & : & \text{sequence}(\text{check}) \\ \text{double-cash-ins} & \equiv & \exists\, c : \text{check. count } c \in \text{cash-ins} > 1 \end{array}$$

Concerning the automated invalidation analysis of a refined policy, we propose action-based verification tools. A powerful formalism like HOL is – unlike FOL – fully capable of modeling datatypes like sequences and even provide some automated support like decision integrated procedures in tools like Isabelle [7]. Appendix A.3 contains a complete Isabelle-formalization of the above described invalidation example.

For insider attacks, we inherently concentrate on organisational security since insiders are by definition part of an organisation. Therefore, when scrutinizing policies for invalidation, we typically consider workflows of organisations. These are more naturally expressed as systems of action sequences not as sets. Due to the expressive power of Isabelle/HOL, we can use the above idea of expressing double cash ins and formalize a refined policy for sequences of checks as follows.

```
Policy_seq l ≡ ∀ x. x mem l ⟶ (count x l = 1 ∧ policy x)
```

The predicate `mem` expresses membership of an element in a list. Note, that the element could occur more than once. However, apart from expressing that the `policy` must hold for all elements `x` we can express that `x` should only occur once in sequence `l`.

Implementing sets as sequences is a classical refinement problem in formal software engineering. In formal specification languages like Z, data refinement has long been identified as a central method for developing specifications into implementations [21]. We use the idea of a coupling invariant that builds the essence of a data refinement by expressing a condition between two layers of a refinement enabling to convey soundness conditions between the two. We call this coupling invariant `policy_refinement`.

```
policy_refinement l ≡ Policy(set l) ⇒ Policy_seq l
```

This invariant expresses that for a sequence of checks `l` the corresponding policy for sequences `Policy_sec` is valid if the abstract set-based `Policy` holds for the set of all elements of sequence `l`. We can now invalidate using an example sequence of two elements with the same cash-in. For this `cash_in_seq` we can show that the coupling invariant does not hold.

```
cash_ins_seq = [Check (Dept (''realestate''))(Recp ''insider'')(Sum 100000),
                Check (Dept (''realestate''))(Recp ''insider'')(Sum 100000)]
⟹  ¬ (policy_refinement cash_ins_seq)"
```

This example of an invalidation can be generalized using the existential quantifier of Higher Order Logic by proving lemma `double_cash_ins_invalidation_by_refinement_gen`.

```
∀ s. finite s ∧ s ≠ {} ⟶ (∃ l. set (l) = s ∧ Policy s ⟶ ¬(policy_refinement l))
```

This final example of multiple cash-ins, illustrates how additional insider attacks may be discovered by applying a refinement step to the specification of the data that is used in the workflow of a system. Only the refinement of the data-centric system specification to sequences enabled expressing a policy that prohibits double cash-ins. Following that, we could proceed with the invalidation to exhibit the insider attack.

Because of the expressivity of its logical language, a tool like Isabelle/HOL is suitable to express intricate concepts like sequence based policies and refinement on them. However, the price to pay is that in comparison to MCMAS the analysis has to be guided by the user. The proofs have to be created in an interactive fashion. For examples, see the proof scripts contained in the code in Appendix A.3.

5 Related Work

Popescu *et al.* [22] provide a policy engine for distributed objects implemented in their system Globe. This work is interesting from our perspective because it addresses the gap between abstract policy specifications and their actual implementation as a service in a distributed object system. In the context of model driven engineering similar goals to ours have been pursued by Mouelhi *et al.* [23] also regarding in particular the relationship between specification and testing presenting a generic security meta-model that can be used for early consistency checks in the security policy. Breaux *et al.* [24] consider semi-formal techniques for analysis and tracing of legal requirements. Gofman *et al.* [25] implement policy checking into a virtual machine mechanism.

Generally, the approach of using invalidation is reminiscent of the counterexample guided abstraction refinement approach in model checking [26] commonly called CEGAR feedback loop. This abstraction process uses an invalidation of an abstraction of a concrete specification to gradually approximate a faithful model checking representation. Conversely, we use the concrete level to invalidate abstract specifications representing a security policy.

The work by Dimkov *et al.* [12] is relevant as it also uses an invalidation procedure to derive attack scenarios. Pieters, Dimkov, and Pavlovic [27] also consider formal techniques for policy alignment to address different levels of abstraction of socio-technical systems. Policy alignment there is a refinement of policies to system representations while policies are seen as theories of first-order logic. These theories contain all behaviours represented as sequences of actions. Policies are defined as 'distinguished' prefix-closed predicates in these theories. Refinement of consistent, i.e., policy-fulfilling specifications, is then implicitly given by the completeness relation amongst first-order theories. Pieters et al. do not use an explicit infrastructure model in their approach [27] but only an abstract formal notion of action sequences to represent systems.

Own previous work relevant for the current paper includes the paper [17] in which a simpler Janitor example has been presented in MCMAS to illustrate the insider problem in a more general setting of formal methods for security. The current paper extends the WRIT workshop paper [28] by giving detailed descriptions of the MCMAS formalization process, a fully fledged Janitor case study, and for the first time the presentation of the road apple example in MCMAS.

6 Conclusion

In this paper, we have presented an approach to supporting invalidating security policies with structural information from organisational models. These two possibilities have been illustrated on examples: the classical janitor example, the road apple attack, and an insider attack case study (DC real estate tax fraud). Techniques for the automated analysis have been introduced and explained on the case studies. For invalidation of policies based on system models we propose the modeling and automated verification with the epistemic logic model checker MCMAS. The two case studies of the Janitor and the road apple have been presented and invalidation properties have revealed insider attacks. For the more data-centric workflow based cases we proposed modeling and interactive proof of invalidation of policies with Isabelle/HOL. The mere logical formalization of a policy and its logical invalidation by negation can already reveal attack possibilities as we illustrated on the DC Fraud example. Two of the actual attacks can be revealed and an additional attack ("bribing the supervisor") is discovered. However, double cash-ins occuring in the real DC Fraud insider attack are not detectable because our set-based model of the workflow is too abstract. Using data refinement techniques, here replacing sets by sequences, enables excluding double cash-ins in the policy. The refinement invariant is used for proving invalidations on example action sequences with double cash-ins.

One issue with our proposition for invalidating policies for insider attacks is the need for a common framework possibly compatible with a widely known policy language like XACML. This should not pose major problems but must be inlined with a series of representative case studies to ensure that the designed common policy language will be sufficiently expressive to cover common scenarios. The technique of invalidation has been demonstrated to be suitable for case studies from insider attacks. We believe that this particular domain profits from the invalidation technique more than general policies since the insider attacks are centered on breaking in from within, *i.e.*, not violating the usual rules.

Acknowledgments

Part of the research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318003 (TREsPASS). This publication reflects only the authors' views and the Union is not liable for any use that may be made of the information contained herein.

We would like to thank the anonymous referees for their constructive comments that helped to improve the paper.

References

[1] C. W. Probst, J. Hunker, D. Gollmann, and M. Bishop, Eds., *Insider Threats in Cybersecurity*. Springer, 2010.

[2] S. Stolfo, S. Bellovin, S. Hershkop, A. Keromytis, S. Sinclair, and S. W. Smith, Eds., *Insider Attack and Cyber Security: Beyond the Hacker*. Springer, 2008.

[3] E. Cole and S. Ring, *Insider Threat: Protecting the Enterprise from Sabotage, Spying, and Theft*. Elsevier, 2006.

[4] B. T. Contos, *Enemy at the Water Cooler*. Elsevier, 2007.

[5] D. M. Cappelli, A. P. Moore, and R. F. Trzeciak, *The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud)*, 1st ed., ser. SEI Series in Software Engineering. Addison-Wesley Professional, February 2012. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321812573

[6] A. Lomuscio, H. Qu, and F. Raimondi, "MCMAS: A model checker for the verification of multi-agent systems," in *Proc. of the 21st International Conference on Computer Aided Verification (CAV'09), Grenoble, France, LNCS*, vol. 5643, June/July 2009, pp. 682–688.

[7] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer-Verlag, 2002, vol. 2283.

[8] S. L. Pfleeger, J. B. Predd, and J. H. C. Bulford, "Insiders behaving badly," *IEEE Transactions on Information Forensics and Security*, vol. 5, no. 1, pp. 169–179, 2010.

[9] C. D. Leonnig and D. Nakamura, "D.C. Tax Scam Total Rises to $20 Million, Officials Say," *Washington Post*, p. A01, 2007.

[10] A. Beautement, M. Sasse, and M. Wonham, "The compliance budget: Managing security behaviour in organisations," in *Proc. of the 2008 Workshop on New Security Paradigms (NSPW'08), Lake Tahoe, California, USA*. ACM, September 2008, pp. 47–58. [Online]. Available: http://dx.doi.org/10.1145/1595676.1595684

[11] C. W. Probst and R. R. Hansen, "An extensible analysable system model," *Information Security Technical Report*, vol. 13, no. 4, pp. 235–246, November 2008.

[12] T. Dimkov, W. Pieters, and P. Hartel, "Portunes: representing attack scenarios spanning through the physical, digital and social domain," in *the 2010 Joint Conference on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'10), Paphos, Cyprus, Revised Selected Papers, LNCS*, vol. 6186. Springer, 2010, pp. 112–129.

[13] S. Stasiukonis, "Social engineering the usb way," 2006, last accessed February 2013. [Online]. Available: http://www.darkreading.com/security/news/208803634/social-engineering-the-usb-way.html

[14] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication," *ACM Transactions on Computer Systems*, vol. 8, pp. 18–36, 1990.

[15] M. Cohen and M. Dam, "A complete axiomatization of knowledge and cryptography," in *Proc. of the 22nd IEEE Symposium on Logic in Computer Science (LICS'07), Wroclaw, Poland*. IEEE, July 2007, pp. 77–88. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/LICS.2007.4

[16] M. Dam, "A little knowledge goes a bit further. invited talk," in *Annual Meeting of Priority Program RS3 – Reliably Secure Software Systems, Karlsruhe, Germany*, 2011.

[17] F. Kammüller, C. Probst, and F. Raimondi, "Application of verification techniques to security – modelchecking insider attacks," in *Advanced Research and Trends in New Technologies, Software, Human-Computer Interaction and Communicability*, F. V. Cipolla-Ficarra, Ed. IGI Global: Hershey, August 2013, pp. 61–70, keynote at conference SETECEC 2012, appears as book chapter.

[18] J. Y. Halpern and V. Weissmann, "Using first-order logic to reason about policies," *ACM Transactions on Information and Systems Security*, vol. 11, no. 4, 2008.

[19] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*, ser. EATCS Texts in Theoretical Computer Science. Springer, 2004.

[20] Inria, "The coq proof assistant," 2014, accessed on 6.6.2014. [Online]. Available: http://coq.inria.fr/

[21] J. He, C. A. R. Hoare, and J. W. Sanders, "Data refinement refined," in *Proc. of the 1986 European Symposium on Programming (ESOP'86), Saarbrücken, Germany, LNCS*, vol. 213. Springer-Verlag, March 1986, pp. 187–196. [Online]. Available: http://dx.doi.org/10.1007/3-540-16442-1_14

[22] B. C. Popescu, B. Crispo, A. S. Tanenbaum, and M. Zeeman, "Enforcing security policies for distributed objects applications," in *The 11th International Workshop on Security Protocols, Cambridge, UK, April 2-4, 2003, Revised Selected Papers, LNCS*, vol. 3364. Springer-Verlag, 2005, pp. 119–130. [Online]. Available: http://dx.doi.org/10.1007/11542322_16

[23] T. Mouelhi, F. Fleurey, B. Baudry, and Y. L. Traon, "A model-based framework for security policy specification, deployment and testing," in *Proc. of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08), Toulouse, France, LNCS*, vol. 5301. Springer-Verlag, September/October 2008, pp. 537–552. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87875-9_38

[24] T. D. Breaux and D. G. Gordon, "Regulatory requirements traceability and analysis using semi-formal specifications," in *Proc. of the19th Working Conference on Requirements Engineering: Foundations for Software*

Quality (REFSQ'13), Essen, Germany, LNCS, vol. 7803. Springer-Verlag, April 2013.

[25] M. I. Gofman, R. Luo, P. Yang, and K. Gopalan, "SPARC: a security and privacy aware virtual machinecheckpointing mechanism," in *Proc. of the 10th Annual ACM Workwhop on Privacy in the Electronic Society (WPES'11), Chicago, Illinois, USA*, October 2011, pp. 115–124. [Online]. Available: http://doi.acm.org/10.1145/2046556.2046571

[26] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proc. of the 12th International Conference on Computer Aided Verification (CAV'00), Chicago, Illinois, USA, LNCS*, vol. 1855. Springer-Verlag, July 2000, pp. 154–169. [Online]. Available: http://dx.doi.org/10.1007/10722167_15

[27] W. Pieters, T. Dimkov, and D. Pavlovic, "Security policy alignment: A formal approach," *IEEE Systems Journal*, vol. 7, no. 2, pp. 275–287, 2013.

[28] F. Kammüller and C. Probst, "Invalidating policies using structural information," in *Proc. of the 2013 IEEE Security and Privacy Workshops (SPW'13), San Francisco, California, USA*. IEEE, May 2013, pp. 76–81, co-located with IEEE CS Security and Privacy 2013.

Author Biography

Florian Kammüller is Senior Lecturer in the Department of Computer Science at Middlesex University London and Privatdozent (honorary professor) at Technische Universitaet Berlin. He holds a Ph.D. from the University of Cambridge on Modular Reasoning in Isabelle and a Habilitation from Technische Universitaet Berlin on Applications of Interactive Theorem Proving in Software Engineering. In his research work that is amply documented in publications in journals and at international conferences he focusses on the support of formal methods with automated verification ranging from bytecode verifiers to UML development and from model checking to constructive type theory. In international collaboration with INRIA Sophia-Antipolis and Microsoft research Cambridge amongst others, he mechanized a calculus and security type system for distributed active objects with Isabelle/HOL and verified secure protocols for the internet, like the secured domain name system DNSsec or social networks.

Christian W. Probst is an Associate Professor in the Department of Applied Mathematics and Computer Science at the Technical University of Denmark, where he works in the section for Language-Based Technologies. The motivation behind Christian's research is to realize systems with guaranteed properties. An important aspect of his work are questions related to safety and security properties, most notably insider threats. He is the creator of ExASyM, the extendable, analysable system model, which supports the identification of insider threats in organisations. Christian has co-organized cross-disciplinary workshops on insider threats and has co-edited a book on the topic.

A Appendix

Appendix A.1 shows the MCMAS specification for the Janitor application mentioned in Section 3: specifications for the agent User A.1.1, the agent Janitor A.1.2, and the final part of an MCMAS file containing the `Evaluation` section defining the attack properties, the initialization of the agents' state variables,

and finally the properties verified by MCMAS (see Appendix A.1.3). Appendix A.1.4 shows the verbose output of MCMAS indicating the attack path and a sample for a False property. Appendix A.2 shows the entire MCMAS file for the roadapple attack where the invalidation state is the world wide web. Appendix A.3 finally, contains the Isabelle-experiment we conducted to support our argument about workflow policy invalidation and refinement.

A.1 Janitor in MCMAS

A.1.1 Specification of agent user

```
Agent User
  -- locations are part of the Agent User and User Janitor's state
Vars:
   initialposition : { outside }; -- The initial state is outside and moves are spelled out
   currentposition : { outside, hall, pc, server }; -- The current position
end Vars
  Actions = { print, movein, move, move1, move2, moveout };
Protocol:
  currentposition = pc or currentposition = server : { move };
  currentposition = outside : { movein };
  currentposition = hall : { move1, move2, moveout };
  currentposition = pc : { print };
end Protocol
Evolution:
   currentposition = hall if (currentposition = outside and Action = movein);
   currentposition = outside if (currentposition = hall and Action = moveout);
   currentposition = pc if (currentposition = hall and Action = move1);
   currentposition = pc if (currentposition = pc and Action = print);
   currentposition = server if (currentposition = hall and Action = move2);
   currentposition = hall if ((currentposition = pc or currentposition = server)
                              and Action = move);
end Evolution
end Agent
```

A.1.2 Janitor specification in MCMAS

```
Agent Janitor
  -- The Janitor
Vars:
   initialposition : { janitor }; -- The initial state is in the janitor room
   currentposition : { outside, hall, server, janitor }; -- The current position
   -- the data is modelled flatly as a boolean flags
   has_secretfile: boolean;
end Vars
  Actions = { pickfromwaste, move, movej, movep, moveout, movein };
Protocol:
  currentposition = janitor or currentposition = server : { move, pickfromwaste };
  currentposition = hall : { movej, movep, moveout };
end Protocol
Evolution:
   currentposition = server if (currentposition = hall and Action = movep);
   currentposition = janitor if (currentposition = hall and Action = movej);
   currentposition = hall if (((currentposition = server or currentposition = janitor)
                               and Action = move)
                              or (currentposition = outside and Action = movein));
```

```
   currentposition = outside if ((currentposition = hall) and Action = moveout);
   has_secretfile = true if (currentposition = server and User.Action = print
                             and  Action = pickfromwaste);
end Evolution
end Agent
```

A.1.3 Janitor proved properties

```
Evaluation
  -- the attack
  janitor_succeeds if Janitor.has_secretfile = true;
  file_outside if (Janitor.has_secretfile = true and Janitor.currentposition = outside);
end Evaluation

InitStates
  -- set the agent's initial states and keys
  (User.initialposition = outside and User.currentposition = outside)
  and
  (Janitor.initialposition = janitor and Janitor.currentposition = janitor)
  and
  (Janitor.has_secretfile = false)
  and
  (User.print_secretfile = false);
end InitStates

Formulae
  -- Janitor can get printout of the secret file; checks True
   EF(janitor_succeeds);
  -- Hence, janitor can carry it outside; checks True
   EF(file_outside);
  -- Consequently attack safety checks False
   AG(!janitor_succeeds);
end Formulae
```

A.1.4 MCMAS sample output for Janitor

```
bash-3.2$ mcmas -c 3 janitor.ispl

************************************************************************
                       MCMAS v1.0.0

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permited by applicable law.

 Please check http://www-lai.doc.ic.ac.uk/mcmas/ for the latest release.
 Report bugs to <mcmas@imperial.ac.uk>
************************************************************************

janitor.ispl has been parsed successfully.
Global syntax checking...
Done
Encoding BDD parameters...
Building partial transition relation...
Building OBDD for initial states...
Building reachable state space...
Checking formulae...
```

```
Verifying properties...
  Formula number 1: (EF janitor_succeeds), is TRUE in the model
.....
----------------------------------------
  Formula number 2: (EF file_outside), is TRUE in the model
  The following is a witness for the formula:
   < 0 1 2 3 4 5 >
  States description:
------------ State: 0 -----------------
Agent Environment
Agent User
  currentposition = outside
  initialposition = outside
Agent Janitor
  currentposition = janitor
  has_secretfile = false
  initialposition = janitor
----------------------------------------
------------ State: 1 -----------------
Agent Environment
Agent User
  currentposition = hall
  initialposition = outside
Agent Janitor
  currentposition = hall
  has_secretfile = false
  initialposition = janitor
----------------------------------------
------------ State: 2 -----------------
Agent Environment
Agent User
  currentposition = pc
  initialposition = outside
Agent Janitor
  currentposition = server
  has_secretfile = false
  initialposition = janitor
----------------------------------------
------------ State: 3 -----------------
Agent Environment
Agent User
  currentposition = pc
  initialposition = outside
Agent Janitor
  currentposition = server
  has_secretfile = true
  initialposition = janitor
----------------------------------------
------------ State: 4 -----------------
Agent Environment
Agent User
  currentposition = hall
  initialposition = outside
Agent Janitor
  currentposition = hall
  has_secretfile = true
  initialposition = janitor
----------------------------------------
------------ State: 5 -----------------
```

```
Agent Environment
Agent User
  currentposition = pc
  initialposition = outside
Agent Janitor
  currentposition = outside
  has_secretfile = true
  initialposition = janitor
----------------------------------------
  Formula number 3: (AG (! janitor_succeeds)), is FALSE in the model
  The following is a counterexample for the formula:
   < 0 >
   < 0 1 2 3 >
  States description:
------------- State: 0 -----------------
Agent Environment
Agent User
  currentposition = outside
  initialposition = outside
Agent Janitor
  currentposition = janitor
  has_secretfile = false
  initialposition = janitor
----------------------------------------
------------- State: 1 -----------------
Agent Environment
Agent User
  currentposition = hall
  initialposition = outside
Agent Janitor
  currentposition = hall
  has_secretfile = false
  initialposition = janitor
----------------------------------------
------------- State: 2 -----------------
Agent Environment
Agent User
  currentposition = pc
  initialposition = outside
Agent Janitor
  currentposition = server
  has_secretfile = false
  initialposition = janitor
----------------------------------------
------------- State: 3 -----------------
Agent Environment
Agent User
  currentposition = pc
  initialposition = outside
Agent Janitor
  currentposition = server
  has_secretfile = true
  initialposition = janitor
----------------------------------------
done, 3 formulae successfully read and checked
execution time = 0
number of reachable states = 32
BDD memory in use = 9057136
```

A.2 Road apple attack in MCMAS

```
Agent User
  -- locations are part of the Agent User and User Janitor's state
Vars:
   initialposition :  outside ; -- The initial state is outside and moves are spelled out
   currentposition :  outside,hall,pc1,pc2 ; -- The current position
end Vars
  Actions =  pick, download, movein, move, move1, move2, moveout ;
Protocol:
  currentposition = pc1 or currentposition = pc2 : move;
  currentposition = pc1 : download, move;
  currentposition = outside : movein,pick;
  currentposition = hall : move1,move2,moveout;
end Protocol
Evolution:
   currentposition = hall if (currentposition = outside and Action = movein);
   currentposition = outside if (currentposition = hall and Action = moveout);
   currentposition = pc1 if (currentposition = hall and Action = move1);
   currentposition = pc2 if (currentposition = hall and Action = move2);
   currentposition = hall if ((currentposition = pc1 or currentposition = pc2) and Action = move);
end Evolution
end Agent


Agent Malex
  -- The Malicious process
Vars:
   initialposition :  outside ; -- The initial state is in the janitor room
   currentposition :  outside, hall, server, pc1, pc2 ; -- The current position
   pickedup: boolean;
   active: boolean;
end Vars
  Actions =  send_secret_data, move, move1, move2, movein, moveout ;
Protocol:
  currentposition = pc2 or currentposition = pc1 : move;
  currentposition = pc1: send_secret_data;
  currentposition = hall : move1,move2,moveout;
  currentposition = outside: movein;
end Protocol
Evolution:
   currentposition = outside if (currentposition = hall and pickedup = true and User.Action = moveout);
   pickedup = true if (currentposition = outside and User.Action = pick);
   currentposition = hall if ((currentposition = outside and pickedup = true and User.Action = movein)
                          or (currentposition = pc1 and pickedup = true and User.Action = move)
                          or (currentposition = pc2 and pickedup = true and User.Action = move));
   currentposition = pc1 if (currentposition = hall and User.Action = move1 and pickedup = true);
   currentposition = pc2 if (currentposition = hall and User.Action = move2 and pickedup = true);
   active = true if (currentposition = pc1 and User.Action = download);
end Evolution
end Agent

Evaluation
  -- the attack
  data_loss if (Malex.active = true);
end Evaluation

InitStates
```

```
  -- set the agent's initial states and keys
  (User.initialposition = outside and User.currentposition = outside)
  and
  (Malex.initialposition = outside and Malex.currentposition = outside)
  and
  (Malex.pickedup = false and Malex.active = false);
end InitStates

Formulae
-- The malicious process can get onto pc1 and sent secret data outside
  EF(data_loss);

end Formulae
```

A.3 DC Fraud in Isabelle

```
theory DCfraud
imports Main
begin
datatype department = Dept string
datatype recipient  = Recp string
datatype sum = Sum nat
datatype check = Check department recipient sum
consts supervise :: check ⇒ bool
consts cash_ins :: check set
consts fake :: check

primrec get_nat_sum :: sum ⇒ nat ("#_")
where  "#(Sum n) = n"
primrec get_dept :: check ⇒ department  ("_.department")
where get_dept (Check d r s) = d
primrec get_recp :: "check ⇒ recipient"("_.recipient")
where get_recp (Check d r s) = r
primrec get_sum :: "check ⇒ sum" ("_.sum")
where get_sum (Check d r s) = s

definition own_dept_exempt :: check ⇒ bool where
"own_dept_exempt x ≡ x.department = Dept(''realestate'')"

definition treshold_exempt :: check ⇒ bool where
"treshold_exempt x ≡ ((#((x).sum)) < 40000)"

definition policy :: check ⇒ bool where
"policy x ≡ own_dept_exempt x ∨ treshold_exempt x ∨ supervise x"

lemma invalidate_one: own_dept_exempt fake ⟹ policy fake
by (simp add: policy_def)

lemma invalidate_two: treshold_exempt fake ⟹ policy fake
by (simp add: policy_def)

lemma invalidate_three: supervise fake  ⟹ policy fake
by (simp add: policy_def)

definition Policy :: check set ⇒ bool where
"Policy l ≡ ∀ x ∈ l. policy x"
```

```
lemma Policy_empty: "Policy {}"
by (simp add: Policy_def)

(* Refinement for Invalidation *)
consts cash_ins_seq:: check list

primrec count :: ['a, 'a list] ⇒ nat where
"count a [] = 0" |
"count a (b # l) = if (a = b) then Suc(count a l) else count a l"

definition Policy_seq :: check list ⇒ bool where
"Policy_seq l ≡ ∀ x. x mem l ⟶ (count x l = 1 ∧ policy x)"

lemma no_zero_count_mem: "count a l > 0 ⟶ a mem l"
by (induct_tac l, simp+)

lemma no_double_cash_ins_in_seq_model: "count c l > 1 ⟹ ¬ (Policy_seq l)"
apply (simp add: Policy_seq_def)
apply (rule_tac x = c in exI)
apply (rule conjI)
apply (simp add: no_zero_count_mem)
apply simp
done

definition policy_refinement:: check list ⇒ bool where
"policy_refinement l ≡ Policy(set l) ⇒ Policy_seq l"

lemma ex_double_cash_ins_invalidation_by_refinement:
"cash_ins_seq = [Check (Dept (''realestate''))(Recp ''insider'')(Sum 100000),
                 Check (Dept (''realestate''))(Recp ''insider'')(Sum 100000)]
⟹  ¬ (policy_refinement cash_ins_seq)"
apply (unfold policy_refinement_def)
apply simp;
apply (rule conjI)
apply (erule ssubst)
apply (simp add: Policy_def policy_def own_dept_exempt_def)
apply (rule no_double_cash_ins_in_seq_model)
apply (erule ssubst)
by simp

lemma Finite_set_list: "finite s ⟹ (∃ l. set l = s)"
apply (erule finite.induct)
apply simp
apply (erule exE)
apply (rule_tac x = "a # l" in exI)
by simp

lemma hd_lem[rule_format]: "l ≠ [] ⟹ hd l mem l"
apply (induct_tac l)
by auto

lemma double_cash_ins_invalidation_by_refinement_gen:
"∀ s. finite s ∧ s ≠ {} ⟶ (∃ l. set (l) = s ∧ Policy s ⟶ ¬(policy_refinement l))"
apply (rule allI, rule impI)
apply (erule conjE)
apply (drule Finite_set_list)
apply (erule exE)
apply (rule_tac x = "hd (l) # l" in exI)
apply (rule impI)
```

```
apply (simp add: policy_refinement_def)
apply (rule_tac c = "hd l" in no_double_cash_ins_in_seq_model)
apply (simp add: no_zero_count_mem_one)
apply (drule sym)
by (simp add: hd_lem)

end
```