Where can an Insider attack?

Christian W. Probst[1], René Rydhof Hansen[2], and Flemming Nielson[1]

[1] Informatics and Mathematical Modelling, Technical University of Denmark**
{probst,nielson}@imm.dtu.dk
[2] Department of Computer Science, University of Copenhagen
rrhansen@diku.dk

Abstract. By definition, an insider has better access, is more trusted, and has better information about internal procedures, high-value targets, and potential weak spots in the security, than an outsider. Consequently, an insider attack has the potential to cause significant, even catastrophic, damage to the targeted organisation. While the problem is well recognised in the security community as well as in law-enforcement and intelligence communities, the main resort still is to audit log files *after the fact*. There has been little research into developing models, automated tools, and techniques for analysing and solving (parts of) the problem.
In this paper we first develop a formal model of systems, that can describe real-world scenarios. These high-level models are then mapped to acKlaim, a process algebra with support for access control, that is used to study and analyse properties of the modelled systems. Our analysis of processes identifies which actions may be performed by whom, at which locations, accessing which data. This allows to compute a superset of audit results—before an incident occurs.

1 Introduction

One of the toughest and most insidious problems in information security, and indeed in security in general, is that of protecting against attacks from an insider. By definition, an insider has better access, is more trusted, and has better information about internal procedures, high-value targets, and potential weak spots in the security. Consequently, an insider attack has the potential to cause significant, even catastrophic, damage to the targeted IT-infrastructure. The problem is well recognised in the security community as well as in law-enforcement and intelligence communities, cf. [1, 14, 6]. In spite of this there has been relatively little focused research into developing models, automated tools, and techniques for analysing and solving (parts of) the problem. The main measure taken still is to audit log files *after* an insider incident has occurred [9].

In this paper we develop a formal model that allows to *formally define* a notion of insider attacks and thereby enables to study systems and analyse

** Part of this work has been supported by the EU research project #016004, *Software Engineering for Service-Oriented Overlay Computers*.

the potential consequences of such an attack. Formal modelling and analysis is increasingly important in a modern computing environment with widely distributed systems, computing grids, and service-oriented architectures, where the line between insider and outsider is more blurred than ever.

With this in mind we have developed a formal model in two parts: an abstract high-level system model based on graphs and a process calculus, called *acKlaim*, providing a formal semantics for the abstract model . As the name suggests, the acKlaim calculus belongs to the Klaim family of process calculi [10] that are all designed around the tuple-space paradigm, making them ideally suited for modelling distributed systems like the interact/cooperate and service-oriented architectures. Specifically, acKlaim is an extension of the μKlaim calculus with access-control primitives. In addition to this formal model we also show how techniques from static program analysis can be applied to automatically compute a sound estimate, i.e., an over-approximation, of the potential consequences of an insider attack. This result has two immediate applications—on the one hand it can be used in designing access controls and assigning security clearances in such a way as to minimise the damage an insider can do. On the other hand, it can direct the auditing process after an insider attack has occurred, by identifying *before* the incident which events should be monitored. The important contribution is to separate the actual threat and attack from reachability. Once we have identified, which places an insider can reach, we can easily put existing models and formalisms on top of our model.

The rest of the paper is structured as follows. In the remainder of this section the terms *insider* and *insider threat* are defined. Section 2 introduces our abstract system model and an example system, and Section 3 defines acKlaim, the process calculus we use to analyse these systems. The analysis itself is introduced in Section 4, followed by a discussion of related work (Section 5). Section 6 concludes our paper and gives an outlook and future work.

1.1 The Insider Problem

Recently, the insider problem has attracted interest by both researchers and agencies. However, most of the work is on detecting insider attacks, modelling the threat itself, and assessing the threat. This section gives an overview of existing work.

Bishop [1] introduces different definitions of the insider threat found in literature. The RAND report [14] defines the problem as "*malevolent actions by an already trusted person with access to sensitive information and information systems*", and the insider is defined as "*someone with access, privilege, or knowledge of information systems and services*". Bishop also cites Patzakis [13] to define the insider as "*anyone operating inside the security perimeter*", thereby contrasting it from *outside* attacks like denial-of-service attacks, which originate from outside the perimeter. Bishop then moves on to define the terms *insider* and *insider threat*:

Definition 1. *(Insider, Insider threat). An* insider with respect to rules R *is a user who may take an action that would violate some set of rules R in the security*

policy, were the user not trusted. The insider is trusted to take the action only when appropriate, as determined by the insider's discretion.
The insider threat *is the threat that an insider may abuse his discretion by taking actions that would violate the security policy when such actions are not warranted.*

Obviously, these definitions are expressive in that they connect actors in a system and their actions to the rules of the security policy. On the other hand, they are rather vague, since in a given system it is usually hard to identify vulnerabilities that might occur based on an insider taking unwarranted actions.

In the rest of the paper we will use Bishop's definitions to analyse abstractions of real systems for potential attacks by insiders. To do so, in the next section we define the abstract model of systems, actors, data, and policies.

2 Modelling Systems

This section introduces our abstract model of systems, which we will analyse in Section 4 to identify potential insider threats. Our system model is at a level that is high enough to allow easy modelling of real-world systems. The system model naturally maps to an abstraction using the process calculus acKlaim (Section 3). Here it is essential that our model is detailed enough to allow expressive analysis results. The abstraction is based on a system consisting of locations and actors. While locations are static, actors can move around in the system. To support these movements, locations can be connected by directed edges, which define freedoms of movements of actors. This section first motivates the choice of our abstraction by an example, and thereafter formally defines the elements.

2.1 Example System

In Figure 1 we show our running example system inspired by [1]. It models part of an environment with physical locations (a server/printer room with a wastebasket, a user office, and a janitor's workshop connected through a hallway), and network locations (two computers connected by a network, and a printer connected to one of them). The access to the server/printer room and the user office is restricted by a cipher lock, and additional by use of a physical master key. The actors in this system are a user and a janitor.

Following Bishop's argumentation, the janitor might pose an insider threat to this system, since he is able to access the server room and pick up printouts from the printer or the wastebasket. We assume a security policy, which allows the janitor access to the server room only in case of fire.

2.2 System Definition

We start with defining the notion of an *infrastructure*, which consists of a set of *locations* and *connections*:

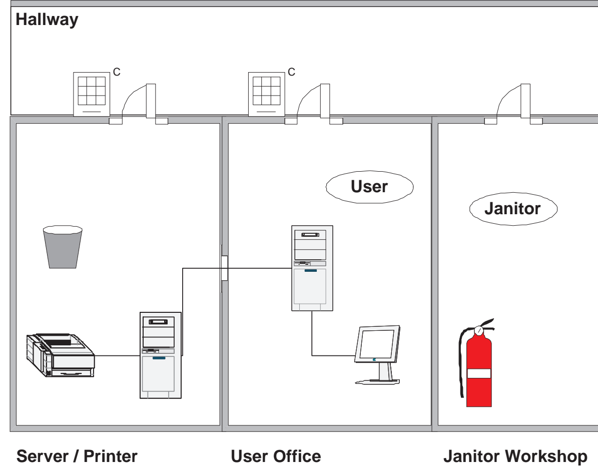

Fig. 1. The example system used to illustrate our approach. The user can use the network connection to print some potentially confidential data in the server/printer room. Depending on the configuration on the cipher locks, the janitor might or might not be able to pick up that print out.

Definition 2. *(Infrastructure, Locations, Connections). An* infrastructure *is a directed graph* $(\mathsf{Loc}, \mathsf{Con})$*, where* Loc *is a set of nodes representing* locations*, and* $\mathsf{Con} \subseteq \mathsf{Loc} \times \mathsf{Loc}$ *is a set of directed* connections *between locations.* $n_d \in \mathsf{Loc}$ *is reachable from* $n_s \in \mathsf{Loc}$*, if there is a path* $\pi = n_0, n_1, n_2, \cdots, n_k$*, with* $k \leq 1$*,* $n_0 = n_s$*,* $n_k = n_d$*, and* $\forall 0 \leq i \leq k-1 : n_i \in \mathsf{Loc} \wedge (n_i, n_{i+1}) \in \mathsf{Con}$.

Next, we define *actors*, which can move in systems by following edges between nodes, and *data*, which actors can produce, pickup, or read. In the example setting, actors would be the user, the janitor, or processes on the computers, whereas data for example would be a printout generated by a program. Usually, actors can only move in a certain domain. In the example system, the user and the janitor can move in the physical locations, but they can only access, e.g., the printer and the waste basket to take items out of them. This is modelled by nodes falling in different domains.

Definition 3. *(Actors, Domains).Let* $\mathcal{I} = (\mathsf{Loc}, \mathsf{Con})$ *be an infrastructure,* Actors *be a set. An* actor $\alpha \in \mathsf{Actors}$ *is an entity that can move in* $\mathcal{I}$*. Let* Dom *be a set of unique domain identifiers. Then* $\mathcal{D} : \mathsf{Loc} \rightarrow \mathsf{Dom}$ *defines the domain* d *for a node* n*, and* $\mathcal{D}^{-1}$ *defines all the nodes that are in a domain.*

Definition 4. *(Data). Let* $\mathcal{I} = (\mathsf{Loc}, \mathsf{Con})$ *be an infrastructure,* Data *be a set of data items, and* $\alpha \in \mathsf{Actors}$ *an actor. A* data item $d \in \mathsf{Data}$ *represents data available in the system. Data can be stored at both locations and actors, and* $\mathcal{K} : (\mathsf{Actors} \cup \mathsf{Loc}) \rightarrow \mathcal{P}(\mathsf{Data})$ *maps an actor or a location to the set of data stored at it.*

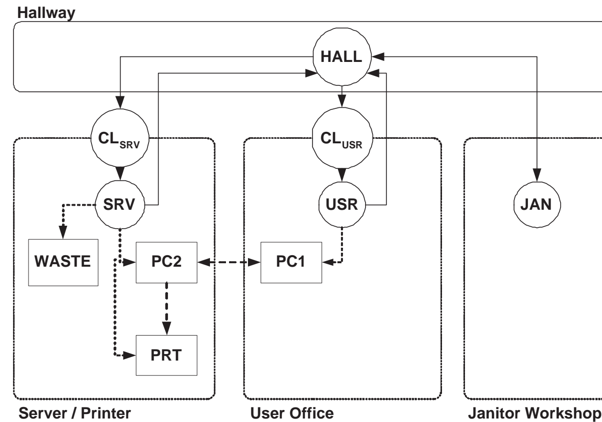

Fig. 2. Abstraction for the example system from Figure 1. The different kinds of arrows indicate how connections can be accessed. The solid lines, e.g., are accessible by actors modelling persons, the dashed lines by processes executing on the network. The dotted lines are special in that they express possible actions of actors.

Finally, we need to model how actors can obtain the right to access locations and data, and how these can decide whether to allow or to deny the access. We associate actors with a set of *capabilities*, and locations and data with a set of *restrictions*. Both restrictions and capabilities can be used to restrain the mobility of actors, by requiring, e.g., a certain key to enter a location, or allowing access only for certain actors or from certain locations. In the example, the code stored in the cipher locks is a restriction, and an actor's knowledge of that code is a capability. Similarly, data items can have access restrictions based on the security classification of the user or based on encryption.

Definition 5. *(Capabilities and Restrictions). Let* $\mathcal{I} = (\mathsf{Loc}, \mathsf{Con})$ *be an infrastructure,* Actors *be a set of actors, and* Data *be a set of data items.* Cap *is a set of* capabilities *and* Res *is a set of restrictions. For each restriction* $r \in \mathsf{Res}$*, the checker* $\Phi_r : \mathsf{Cap} \to \{\mathtt{t}rue, \mathtt{f}alse\}$ *checks whether the capability matches the restriction or not.* $\mathcal{C} : \mathsf{Actors} \to \mathcal{P}(\mathsf{Cap})$ *maps each actor to a set of* capabilities*, and* $\mathcal{R} : (\mathsf{Loc} \cup \mathsf{Data}) \to \mathcal{P}(\mathsf{Res})$ *maps each location and data item to a set of* restrictions.

Figure 2 shows the modelling for the example system from Figure 1. Locations are the rooms and cipher locks (circles), and the computers, the printer, and the wastebasket (squares). The different kinds of arrows indicate how connections can be accessed. The solid lines, e.g., are accessible by actors modelling persons, the dashed lines by processes executing on the network. The dotted lines are special in that they express possible *actions* of actors. An actor at the server location, e.g., can access the wastebasket. Finally, we combine the above elements to a system:

Definition 6. *(System). Let* $\mathcal{I} = (\mathsf{Loc}, \mathsf{Con})$ *be an infrastructure,* Actors *a set of actors in* $\mathcal{I}$*,* Data *a set of data items,* Cap *a set of capabilities,* Res *a set of restrictions,* $\mathcal{C} : \mathsf{Actors} \rightarrow \mathcal{P}(\mathsf{Cap})$ *and* $\mathcal{R} : (\mathsf{Loc} \cup \mathsf{Data}) \rightarrow \mathcal{P}(\mathsf{Res})$ *maps from actors and location and data, respectively, to restrictions and capabilities, respectively, and for each restriction* r*, let* $\Phi_r : \mathsf{Cap} \rightarrow \{\mathtt{true}, \mathtt{false}\}$ *be a checker. Then we call* $\mathcal{S} = \langle \mathcal{I}, \mathsf{Actors}, \mathsf{Data}, \mathcal{C}, \mathcal{R}, \Phi \rangle$ *a system.*

3 The acKlaim Calculus

In this section we present the process calculus that provides the formal underpinning for the system model presented in the previous section. The calculus, called *acKlaim*, belongs to the Klaim family of process calculi [10]; it is a variation of the μKlaim calculus enhanced with access control primitives and equipped with a *reference monitor semantics* (inspired by [8]) that ensures compliance with the system's access policy. In addition to providing a convenient and well-understood formal framework, the use of a process calculus also enables us to apply a range of tools and techniques originally developed in a programming language context. In particular it facilitates the use of *static analysis* to compute a sound approximation of the consequences of an insider attack.

3.1 Access Policies

We start by defining the *access policies* that are enforced by the reference monitor. Access policies come in three varieties: access can be granted based on the location it is coming from, based on the actor that is performing the access, or based on the knowledge of the secret, e.g., a key in (a)symmetric encryption. In

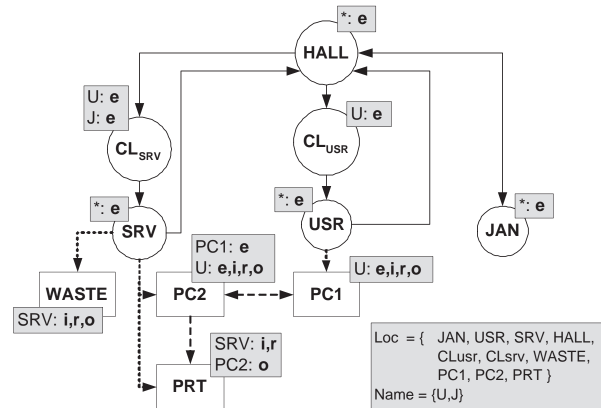

Fig. 3. The abstracted example system from Figure 2, extended with policy annotations. There are two actors, janitor J and user U, who, e.g., have different access rights to the user office and the server room.

the above system model (Definition 5), these are modelled by capabilities and restrictions.

$$\begin{aligned}\pi \subseteq \mathsf{AccMode} &= \{\mathbf{i},\mathbf{r},\mathbf{o},\mathbf{e},\mathbf{n}\}\\ \kappa \subseteq \mathsf{Keys} &= \{\textit{unique key identifiers}\}\\ \delta \in \mathsf{Policy} &= (\mathsf{Loc} \cup \mathsf{Name} \cup \mathsf{Keys} \cup \{\star\}) \to \mathcal{P}(\mathsf{AccMode})\end{aligned}$$

The access modes $\mathbf{i},\mathbf{r},\mathbf{o},\mathbf{e},\mathbf{n}$ correspond to *destructively read a tuple*, *non-destructively read a tuple*, *output a tuple*, *remote evaluation*, and *create new location* respectively. These modes reflect the underlying reference monitor semantics and are explained in detail below. The special element $\star$ allows to specify a set of access modes that are allowed by default. The separation of *location* and *names* is artificial in that both sets simply contain unique identifiers. They are separated to stress the distinction between locations as part of the infrastructure and actors that are moving around in the infrastructure.

The elements of the domain Keys are keys used, e.g., for symmetric or asymmetric encryption. We assume that each key uniquely maps to the method used to check it (a checker Φ_r).

Intuitively, every *locality* in the system defines an access policy that specifies how other localities and actors are allowed to access and interact with it. This approach affords fine-grained control for individual localities over both *who* is allowed access and *how*. Semantically the access control model is formalised by a *reference monitor* embedded in the operational semantics for the calculus. The reference monitor verifies that every access to a locality is in accordance with that locality's access policy.

We use the function *grant*to decide whether an actor n at location l knowing keys κ should be allowed to perform an action a on the location l' (Figure 4).

$$\text{grant} : \mathsf{Names} \times \mathsf{Loc} \times \mathsf{Keys} \times \mathsf{AccMode} \times \mathsf{Loc} \to \{\mathtt{true}, \mathtt{false}\}$$

$$\text{grant}(n, l, \kappa, a, l') = \begin{cases} \mathtt{true} & \text{if } a \in \delta_{l'}(n) \vee a \in \delta_{l'}(l) \vee \exists k \in \kappa : a \in \delta_{l'}(k) \\ \mathtt{false} & \text{otherwise} \end{cases}$$

$$\frac{l = t}{\langle \mathcal{I}, n, \kappa \rangle \succ (l, t)} \qquad \frac{\exists (l, l') \in \mathsf{Con} : \text{grant}(n, l, \kappa, \mathbf{e}, l') \wedge \langle \mathcal{I}, n, \kappa \rangle \succ (l', t)}{\langle \mathcal{I}, n, \kappa \rangle \succ (l, t)}$$

$$\frac{\text{grant}(n, l, \kappa, a, t) \wedge \langle \mathcal{I}, n, \kappa \rangle \succ (l, t)}{\langle \mathcal{I}, n, \kappa \rangle \rightsquigarrow (l, t, a)}$$

Fig. 4. Function *grant* (upper part) checks whether an actor n at location l knowing keys κ should be allowed to perform an action a on the location l' based on the location it is at, its name, or a key it knows. The judgement $\succ$ (lower part) decides whether an actor n at location s can reach location t based on the edges present in the infrastructure $\mathcal{I}$, by testing $\langle \mathcal{I}, n, \kappa \rangle \succ (s, t)$. Finally, the judgement $\rightsquigarrow$ uses *grant* and judgement $\succ$ to test whether n is allowed to execute action a at location t.

$\ell ::= l$	locality	$N ::= l ::^{\delta} [P]^{\langle n,\kappa \rangle}$	single node
$\mid u$	locality variable	$\mid l ::^{\delta} \langle et \rangle$	located tuple
		$\mid N_1 \parallel N_2$	net composition
$P ::= \mathbf{nil}$	null process	$a ::= \mathbf{out}\,(t)\,@\ell$	output
$\mid a.P$	action prefixing	$\mid \mathbf{in}\,(T)\,@\ell$	input
$\mid P_1 \mid P_2$	parallel composition	$\mid \mathbf{read}\,(T)\,@\ell$	read
$\mid A$	process invocation	$\mid \mathbf{eval}\,(P)\,@\ell$	migration
		$\mid \mathbf{newloc}(u^{\pi} : \delta)$	creation

Fig. 5. Syntax of nets, processes, and actions.

$T ::= F \mid F, T$	templates	$et ::= ef \mid ef, et$	evaluated tuple
$F ::= f \mid !x \mid !u$	template fields	$ef ::= V \mid l$	evaluated tuple field
$t ::= f \mid f, t$	tuples	$e ::= V \mid x \mid \ldots$	expressions
$f ::= e \mid l \mid u$	tuple fields		

Fig. 6. Syntax for tuples and templates.

Additionally, access policies can be defined for every *data item*. In this case, only the subset $\{\mathbf{i}, \mathbf{r}\}$ of access modes can be used for name- or location based specification, as well as keys specifying how the data item has been encrypted.

3.2 Syntax and Semantics

The Klaim family of calculi, including acKlaim, are motivated by and designed around the *tuple space* paradigm in which a system consists of a set of distributed nodes that interact and communicate through shared tuple spaces by reading and writing tuples. Remote evaluation of processes is used to model mobility.

The acKlaim calculus, like other members of the Klaim family, consists of three layers: nets, processes, and actions. Nets give the overall structure where tuple spaces and processes are located; processes execute by performing actions. The syntax is shown in Figure 5 and Figure 6. The main difference to standard Klaim calculi is, that processes are annotated with a name, in order to allow modelling of actors moving in a system, and a set of keys to model the capabilities they have.

The semantics for acKlaim (Figure 7) is specified as a small step operational semantics and follows the semantics of μKlaim quite closely. A process is either comprised of subprocesses composed in parallel, an action (or a sequence of actions) to be executed, the nil-process, i.e., the process that does nothing, or it can be a recursive invocation through a place-holder variable explicitly defined by equation. The **out** action outputs a tuple to the specified tuple space; the **in** and **read** actions read a tuple from the specified tuple space in a destructive/non-destructive manner, respectively. When reading from a tuple space, using either the **in** or the **read** action, only tuples that match the input template (see Fig-

$$\frac{[\![t]\!] = et \qquad \boxed{\langle \mathcal{I}, n, \kappa \rangle \rightsquigarrow (l, l', \mathbf{o})}}{l ::^{\delta} [\mathbf{out}\,(t)\,@l'.P]^{\langle n,\kappa \rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa' \rangle} \rightarrowtail_{\mathcal{I}} l ::^{\delta} [P]^{\langle n,\kappa \rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa' \rangle} \parallel l' ::^{\delta'} \langle et \rangle}$$

$$\frac{\mathrm{match}([\![T]\!], et) = \sigma \qquad \boxed{\langle \mathcal{I}, n, \kappa \rangle \rightsquigarrow (l, l', \mathbf{i})}}{l ::^{\delta} [\mathbf{in}\,(T)\,@l'.P]^{\langle n,\kappa \rangle} \parallel l' ::^{\delta'} \langle et \rangle \rightarrowtail_{\mathcal{I}} l ::^{\delta} [P\sigma]^{\langle n,\kappa \rangle} \parallel l' ::^{\delta'} \mathbf{nil}}$$

$$\frac{\mathrm{match}([\![T]\!], et) = \sigma \qquad \boxed{\langle \mathcal{I}, n, \kappa \rangle \rightsquigarrow (l, l', \mathbf{r})}}{l ::^{\delta} [\mathbf{read}\,(T)\,@l'.P]^{\langle n,\kappa \rangle} \parallel l' ::^{\delta'} \langle et \rangle \rightarrowtail_{\mathcal{I}} l ::^{\delta} [P\sigma]^{\langle n,\kappa \rangle} \parallel l' ::^{\delta'} \langle et \rangle}$$

$$\frac{\boxed{\langle \mathcal{I}, n, \kappa \rangle \rightsquigarrow (l, l', \mathbf{e})}}{l ::^{\delta} [\mathbf{eval}\,(Q)\,@l'.P]^{\langle n,\kappa \rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa' \rangle} \rightarrowtail_{\mathcal{I}} l ::^{\delta} [P]^{\langle n,\kappa \rangle} \parallel l' ::^{\delta'} [Q]^{\langle n,\kappa \rangle} \parallel l' ::^{\delta'} [P']^{\langle n',\kappa' \rangle}}$$

$$\frac{l' \notin L \qquad \lfloor l' \rfloor = \lfloor u \rfloor}{L \vdash l ::^{\delta} [\mathbf{newloc}(u^{\pi} : \delta').P]^{\langle n,\kappa \rangle} \rightarrowtail_{\mathcal{I}} L \cup \{l'\} \vdash l ::^{\delta[l' \mapsto \pi]} [P[l'/u]]^{\langle n,\kappa \rangle} \parallel l' ::^{\delta'[l'/u]} [\mathbf{nil}]^{\langle n,\kappa \rangle}}$$

$$\frac{L \vdash N_1 \rightarrowtail_{\mathcal{I}} L' \vdash N_1'}{L \vdash N_1 \parallel N_2 \rightarrowtail_{\mathcal{I}} L' \vdash N_1' \parallel N_2} \qquad \frac{N \equiv N_1 \qquad L \vdash N_1 \rightarrowtail_{\mathcal{I}} L' \vdash N_2 \qquad N_2 \equiv N'}{L \vdash N \rightarrowtail_{\mathcal{I}} L' \vdash N'}$$

Fig. 7. Operational semantics for acKlaim. The semantics is annotated with the spatial structure $\mathcal{I}$ of the underlying physical system. We omit the structure wherever it is clear from context or is not needed. The boxes contain the reference monitor functionality, that uses the structure $\mathcal{I}$ to verify that an intended action is allowable.

$$\mathrm{match}(V, V) = \epsilon \quad \mathrm{match}(!x, V) = [V/x] \qquad \mathrm{match}(l, l) = \epsilon \quad \mathrm{match}(!u, l') = [l'/u]$$

$$\frac{\mathrm{match}(F, ef) = \sigma_1 \qquad \mathrm{match}(T, et) = \sigma_2}{\mathrm{match}((F, T), (ef, et)) = \sigma_1 \circ \sigma_2}$$

Fig. 8. Semantics for template matching.

ure 6) are read. This pattern matching is formalised in Figure 8. The **eval** action implements remote process evaluation, and the **newloc** action creates a new location, subject to a specified access policy. While locations representing physical structures usually would be static in our system view, **newloc** can be used to model, e.g., the spawning of processes in computer systems. Note that the semantic rule for the **newloc** action is restricted through the use of *canonical names*; these give a convenient way for the control flow analysis (cf. Section 4) to handle the potentially infinite number of localities arising from unbounded use of **newloc**. These will be explained in detail in Section 4. As is common for process calculi, the operational semantics is defined with respect to a built-in structural congruence on nets and processes. This simplifies presentation and reasoning about processes. The congruence is shown in Figure 9.

In addition to the features of the standard semantics of μKlaim, we add the *spatial* structure of the system to the semantics of acKlaim. This structure is

used to limit how access to other locations is granted. The system component $\mathcal{S}$ is, among others, represented by a graph $\mathcal{I}$ as specified in Definition 6. The reference monitor passes $\mathcal{I}$ to the judgement $\succ$ (Figure 4) to check whether there is a path from a process's current location and the target location of the action. The reference monitor is formalised as additional premises of the reduction rules, shown as boxes in Figure 7.

3.3 The Example Revisited

We now use acKlaim to model the system as specified in Figure 3, resulting in the acKlaim program in Figure 10. The system property we are most interested in is the spatial structure of the system, therefore most locations run either the **nil** process or have an empty tuple space, if their location does not allow any actor to execute processes at them. The user's office and the janitor's workshop contain process variables, that can be used to plug in and analyse arbitrary processes for these actors.

4 Analysing the System Abstraction

In this section we describe several analyses that we perform on the infrastructure underlying a system as well as on possible actors in the system. The first analysis (Section 4.1) determines, which locations in a system an actor with name n and keys κ can reach from location l—either directly or by performing an action on them. With respect to an insider threat this allows to determine which locations an insider can reach and which data he can potentially access.

$$
\begin{array}{ll}
N_1 \parallel N_2 \equiv N_2 \parallel N_1 & (N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3) \\
l ::^{\delta} [P]^{\langle n,\kappa\rangle} \equiv l ::^{\delta} [(P \mid \mathbf{nil})]^{\langle n,\kappa\rangle} & \\
l ::^{\delta} [A]^{\langle n,\kappa\rangle} \equiv l ::^{\delta} [P]^{\langle n,\kappa\rangle} \quad \text{if } A \triangleq P & \\
l ::^{\delta} [(P_1 \mid P_2)]^{\langle n,\kappa\rangle} \equiv l ::^{\delta} [P_1]^{\langle n,\kappa\rangle} \parallel l ::^{\delta} [P_2]^{\langle n,\kappa\rangle} &
\end{array}
$$

Fig. 9. Structural congruence on nets and processes.

$$
\begin{array}{lll}
\mathrm{HALL} ::^{\langle \star \mapsto \mathbf{e} \rangle} \mathbf{nil} & \parallel \mathrm{USR} ::^{\langle \star \mapsto \mathbf{e} \rangle} U & \parallel \mathrm{JAN} ::^{\langle \star \mapsto \mathbf{e} \rangle} J \parallel \\
\mathrm{CL}_{\mathrm{USR}} ::^{\langle \mathrm{U} \mapsto \mathbf{e} \rangle} \mathbf{nil} & \parallel \mathrm{PC1} ::^{\langle \mathrm{U} \mapsto \mathbf{e},\mathbf{i},\mathbf{r},\mathbf{o} \rangle} \mathbf{nil} \parallel & \\
\mathrm{CL}_{\mathrm{SRV}} ::^{\langle \mathrm{U} \mapsto \mathbf{e}, \mathrm{J} \mapsto \mathbf{e} \rangle} \mathbf{nil} & \parallel \mathrm{SRV} ::^{\langle \star \mapsto \mathbf{e} \rangle} \mathbf{nil} & \parallel \mathrm{WASTE} ::^{\langle \mathrm{SRV} \mapsto \mathbf{i},\mathbf{o},\mathbf{r} \rangle} \langle\rangle \parallel \\
\mathrm{PC2} ::^{\langle \mathrm{PC1} \mapsto \mathbf{e}, \mathrm{U} \mapsto \mathbf{e},\mathbf{i},\mathbf{r},\mathbf{o} \rangle} \mathbf{nil} & \parallel \mathrm{PRT} ::^{\langle \mathrm{SRV} \mapsto \mathbf{i},\mathbf{r}, \mathrm{PC2} \mapsto \mathbf{o} \rangle} \langle\rangle &
\end{array}
$$

Fig. 10. The example system translated into acKlaim. The two process variables J and U can be instantiated to hold actual process definitions. The system property we are most interested in is the spatial structure of the system, therefore most locations run either the **nil** process or have an empty tuple space, if their location does not allow any actor to execute processes at them

This analysis can be compared to a before-the-fact system analysis to identify possible vulnerabilities and actions that an audit should check for.

The second analysis (Section 4.2) is a control-flow analysis of actors in a system. It determines, which locations a specific process may reach, which actions it may execute, and which data it may read. This can be compared to an after-the-fact audit of log files.

4.1 Attack Points

In identifying potential insider attacks in a system, it is important to understand, which locations in the system an insider can actually reach. This reachability problem comes in two variations—first we analyse the system *locally* for a specific actor located at a specific location. Then, we put this information together to identify all system-wide locations in the system that an actor possibly can reach. Finally, the result of the reachability analyses can be used in computing which data an actor may access on system locations, by evaluating which actions he can execute from the locations he can reach.

Given that this analysis is very similar to a reachability analysis, we only sketch how it works. Figure 11 shows the pseudo code specification for both reachability analyses and the global data analysis, parametrised in the system structure, the name n of the actor, and the set of keys κ that the actor knows.

For the example system from Figure 3, the analysis finds out, that the an actor with name J and an empty set of keys can reach the location SRV and can therefore execute the **read** action on both the waste basket and the printer,

$$
\begin{array}{l}
\text{checkloc} : \mathsf{Names} \times \mathsf{Loc} \times \mathsf{Keys} \times (\mathsf{Con} \times \mathsf{Loc}) \rightarrow \mathcal{P}(\mathsf{Loc}) \\
\quad \text{for all } (l, l') \in \mathsf{Con} \\
\quad\quad \text{if } \langle \mathcal{I}, n, \kappa \rangle \succ (l, l') \vee \text{grant}(n, l, \kappa, \mathbf{e}, l') \\
\quad\quad\quad \text{return } \{l'\} \cup \text{checkloc}(n, l', \kappa, \mathcal{I})
\end{array}
$$

$$
\begin{array}{l}
\text{checksys} : \mathsf{Names} \times \mathsf{Keys} \times (\mathsf{Con} \times \mathsf{Loc}) \rightarrow \mathcal{P}(\mathsf{Loc}) \\
\quad \text{checksys}(n, \kappa, \mathcal{I}) = \bigcup_{l \in \mathsf{Loc}} \text{checkloc}(n, l, \kappa, \mathcal{I})
\end{array}
$$

$$
\begin{array}{l}
\text{checkdata} : \mathsf{Names} \times \mathsf{Keys} \times (\mathsf{Con} \times \mathsf{Loc}) \rightarrow \mathcal{P}(\mathsf{AccMode} \times \mathsf{Loc}) \\
\quad \text{checkdata}(n, \kappa, \mathcal{I}) = \bigcup_{l \in \text{checkloc}(n, l, \kappa, \mathcal{I})} \{(a, l) | \exists a \in \mathsf{AccMode}, (l, l') \in \mathsf{Con} : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{grant}(n, l, \kappa, a, l')\}
\end{array}
$$

Fig. 11. Analysing a given system for attack points. The local analysis (upper part) takes a name n, a location l and a key set κ, and returns the set of all locations that n could possibly reach using κ. The system-wide analysis (middle part) puts these together to obtain the global view, by executing the local analysis on all locations in the system. Finally, the data analysis uses the system-wide analysis to compute at which locations the actor may invoke which actions, allowing to identify which data he may possibly access. A local data-analysis similar to the upper part is defined analogous.

possibly accessing confidential data printed or trashed. While this is obvious for the simplistic example system, those properties are usually not easily spotted in complex systems.

4.2 Control-flow Analysis

While the reachability analysis defined in the previous section is easily computed and verified, it also is highly imprecise in that it does not take into account the actual actions executed in an attack. As described before, the reachability analysis can be used in identifying vulnerable locations that might have to be put under special scrutiny.

In this section we specify a static *control flow analysis* for the acKlaim calculus. The control flow analysis computes a conservative approximation of all the possible flows into and out of all the tuple spaces in the system. The analysis is specified in the Flow Logic framework [11], which is a specification-oriented framework that allows "rapid development" of analyses. An analysis is specified through a number of *judgements* that each determine whether or not a particular analysis estimate correctly describes all configurations that are reachable from the initial state. Concretely we define three judgements: for nets, processes, and actions respectively. The definitions are shown in Figure 12.

Information is collected by the analysis in two components: $\hat{T}$ and $\hat{\sigma}$. The former records for every tuple space (an over-approximation of) the set of tuples possibly located in that tuple space at any point in time. The latter component tracks the possible values that variables may be bound to during execution, i.e., this component acts as an abstract environment.

We briefly mention a technical issue before continuing with specification of the analysis: the handling of dynamic creation of locations. In order to avoid having the analysis keep track of a potentially infinite number of locations we define

$$
\begin{array}{lll}
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{N}} l ::^{\delta} [P]^{\langle n,\kappa\rangle} & \text{iff} & (\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{P}}^{\lfloor l\rfloor,n,\kappa} P \\
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{N}} l ::^{\delta} \langle et\rangle & \text{iff} & \langle et\rangle \in \hat{T}(\lfloor l\rfloor) \\
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{N}} N_1 \parallel N_2 & \text{iff} & (\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{N}} N_1 \wedge (\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{N}} N_2 \\
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{P}}^{l,n,\kappa} \mathbf{nil} & \text{iff} & \mathit{true} \\
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{P}}^{l,n,\kappa} P_1 \mid P_2 & \text{iff} & (\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{P}}^{l,n,\kappa} P_1 \wedge (\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{P}}^{l,n,\kappa} P_2 \\
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{P}}^{l,n,\kappa} A & \text{iff} & (\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{P}}^{l,n,\kappa} P \qquad \text{if } A \triangleq P \\
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{P}}^{l,n,\kappa} a.P & \text{iff} & (\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{A}}^{l,n,\kappa} a \wedge (\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{P}}^{l,n,\kappa} P \\
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{A}}^{l,n,\kappa} \mathbf{out}\,(t)\,@\ell' & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell')\colon (\langle \mathcal{I},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathbf{o}) \Rightarrow \hat{\sigma}[\![t]\!] \subseteq \hat{T}(\hat{l})) \\
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{A}}^{l,n,\kappa} \mathbf{in}\,(T)\,@\ell' & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell')\colon (\langle \mathcal{I},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathbf{i}) \Rightarrow \hat{\sigma} \models_1 T : \hat{T}(\hat{l}) \triangleright \hat{W}_{\bullet}) \\
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{A}}^{l,n,\kappa} \mathbf{read}\,(T)\,@\ell' & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell')\colon (\langle \mathcal{I},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathbf{r}) \Rightarrow \hat{\sigma} \models_1 T : \hat{T}(\hat{l}) \triangleright \hat{W}_{\bullet}) \\
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{A}}^{l,n,\kappa} \mathbf{eval}\,(Q)\,@\ell' & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell') : (\langle \mathcal{I},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathbf{e}) \Rightarrow (\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{P}}^{\hat{l},n,\kappa} Q) \\
(\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{A}}^{l,n,\kappa} \mathbf{newloc}(u^{\pi} : \delta) & \text{iff} & \{\lfloor u\rfloor\} \subseteq \hat{\sigma}(\lfloor u\rfloor)
\end{array}
$$

Fig. 12. Flow Logic specification for control flow analysis of acKlaim.

$$
\begin{array}{lll}
\hat{\sigma} \models_i \epsilon : \hat{V}_\circ \triangleright \hat{V}_\bullet & \text{iff} & \{\hat{et} \in \hat{V}_\circ \,|\, |\hat{et}| = i\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i V, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \{\hat{et} \in \hat{V}_\circ \,|\, \pi_i(\hat{et}) = V\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i l, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \{\hat{et} \in \hat{V}_\circ \,|\, \pi_i(\hat{et}) = V\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i x, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \{\hat{et} \in \hat{V}_\circ \,|\, \pi_i(\hat{et}) \in \hat{\sigma}(x)\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i u, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \{\hat{et} \in \hat{V}_\circ \,|\, \pi_i(\hat{et}) \in \hat{\sigma}(u)\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i !x, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \hat{V}_\circ \sqsubseteq \hat{V}_\bullet \wedge \pi_i(\hat{W}_\bullet) \sqsubseteq \hat{\sigma}(x) \\
\hat{\sigma} \models_i !u, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \wedge \hat{V}_\circ \sqsubseteq \hat{V}_\bullet \wedge \pi_i(\hat{W}_\bullet) \sqsubseteq \hat{\sigma}(u)
\end{array}
$$

Fig. 13. Flow Logic specification for pattern match analysis.

and use so-called *canonical names* that divides all concrete location names and location variables into equivalence classes in such a way that all (new) location names generated at the same program point belong to the same equivalence class and thus share the same canonical name. The canonical name (equivalence class) of a location or location variable, ℓ, is written $\lfloor \ell \rfloor$. In the interest of legibility we use a *unique representative* for each equivalence class and thereby dispense with the $\lfloor \cdot \rfloor$ notation whenever possible. We avoid possible inconsistencies in the security policy for two locations with the same canonical names we only consider policies that are *compatible* with the choice of canonical names; a policy is compatible if and only if $\lfloor \ell_1 \rfloor = \lfloor \ell_2 \rfloor \Rightarrow \delta(\ell_1) = \delta(\ell_2)$. This implies that the policy assigns the exact same set of capabilities to all locations with the same canonical name. Throughout this paper we assume that policies are compatible with the chosen canonical names.

A separate Flow Logic specification, shown in Figure 13, is developed in order to track the pattern matching performed by both the input actions.

Having specified the analysis it remains to be shown that the information computed by the analysis is correct. In the Flow Logic framework this is usually done by establishing a *subject reduction* property for the analysis:

Theorem 1 (Subject Reduction). *If* $(\hat{T}, \hat{\sigma}, \mathcal{I}) \models_N N$ *and* $L \vdash N \rightarrowtail_{\mathcal{I}} L' \vdash N'$ *then* $(\hat{T}, \hat{\sigma}, \mathcal{I}) \models_N N'$.

Proof. (Sketch) By induction on the structure of $L \vdash N \rightarrowtail_{\mathcal{I}} L' \vdash N'$ and using auxiliary results for the other judgements.

Now we return to the abstract system model for our example (Figure 10). To analyse it, we replace the two process variables J and U with processes as specified in Figure 14. The janitor process J moves from the janitor's workshop location JAN to the server room SRV where he picks up a the review for the insider paper from the printer (**in** ("review", "insiderpaper", $!r$) @PRT). The user U prints the review from PC1 via the print server PC2. The two interesting locations for the control-flow analysis are JAN and USR, where processes J and U are plugged in, respectively. When analysing U, the analysis starts by analysing **eval** (**out** ($\cdots$) @PRT) @PC2, resulting in an analysis of the action **out** (("review", "insiderpaper", "accept")) @PRT. As specified in Figure 12, this results in the tuple ("review", "insiderpaper", "accept") being stored in $\hat{T}$(PRT),

$$J \triangleq \textbf{eval}\,(\textbf{in}\,((\text{"review"}, \text{"insiderpaper"}, !r))\ @\text{PRT})\ @\text{SRV}$$

$$U \triangleq \textbf{eval}\,(\textbf{eval}\,(\textbf{out}\,((\text{"review"}, \text{"insiderpaper"}, \text{"accept"}))\ @\text{PRT})\ @\text{PC2})\ @\text{PC1}$$

Fig. 14. Processes for the janitor and the user analysed in the abstract system model from Figure 10.

representing the fact that the user started a print job at the print server, and the resulting document ended up in the printer. The analysis of J results in analysing **in** ((”review”, ”insiderpaper”, $!r$)) @PRT, which tries to read a tuple matching the first two components from the tuple space at locations PRT. Since that is available after U has been analysed[3], the local variable r contains the string ”accept”, even though the janitor might not be authorised to access this data. Note that for sake of simplicity we do not have added security classifications to our model, but any mechanism could easily be added on top of the model and the analysis.

5 Related Work

Recently, insiders and the insider threat [14, 15, 3, 2] have attracted increased attention due to the potential damage an insider can cause. Bishop [1] gives an overview of different definitions and provides unified definition, which is the basis for our work. By separating the reachability analysis from the actual threat, we are able to easily model other definitions of the insider threat, or insiders that are more capable.

While the aforementioned papers discuss the insider problem, only very little work can be found on the static analysis of system models with respect to a potential insider threat. Chinchani et al. [4] describe a modelling methodology which captures several aspects of the insider threat. Their model is also based on graphs, but the main purpose of their approach is to reveal possible attack strategies of an insider. They do so by modelling the system as a key challenge graph, where nodes represent physical entities that store some information or capability. Protections like, e.g., the cipher locks in our example, are modelled as key challenges. For legitimate accesses these challenges incur no costs, while for illegitimate accesses they incur a higher cost representing the necessary “work” to guess or derive the key. The difference to our approach is that they start with a set of target nodes and compute an attack that compromises these nodes. However, it is mostly unclear how the cost of breaking a key challenge is determined. We are currently working on incorporating probabilities into our model to express the likelihood of a certain capability being acquired by a malicious insider. It might be interesting to incorporate this into Chinchani’s approach and to execute our analysis on their graphs to compare these two approaches.

3

In a more general setting, fault trees have been used for analysing for system failures [7]. However, these have not been used to model attacks, but to compute the chance of combinations of faults to occur. Beside these, graphs have been used in different settings to analyse attacks on networks. Examples include privilege graphs [5, 12] and attack graphs [16]. The major difference to our work is the level of detail in modelling static and dynamic properties of the system, and the ability to analyse the dynamic behaviour of actors moving in the abstract system.

6 Conclusion and Future Work

One of the most insidious problems in information security is that of protecting against attacks from an insider. Even though the problem is well recognised in the security community as well as in law-enforcement and intelligence communities, there has been relatively little focused research into developing models and techniques for analysing and solving (parts of) the problem. The main measure taken still is to audit log files *after* an insider incident has occurred.

We have presented a formal model that allows to *formally define* a notion of insider attacks and thereby enables to study systems and analyse the potential consequences of such an attack. Formal modelling and analysis is increasingly important in a modern computing environment with widely distributed systems, computing grids, and service-oriented architectures, where the line between insider and outsider is more blurred than ever.

The two components of our model—an abstract high-level system model based on graphs and the process calculus *acKlaim*—are expressive enough to allow easy modelling of real-world systems, and detailed enough to allow expressive analysis results. On the system model, we use reachability analysis to identify possible vulnerabilities and actions that an audit should check for, by computing locations that actors in the system can reach and/or access—independent of the actual actions they perform. On the abstract acKlaim model we perform a control-flow analysis of specific processes/actors to determine which locations a specific process may reach, which actions it may execute, and which data it may read. This can be compared to an after-the-fact audit of log files. To the best of our knowledge this is the first attempt in applying static analysis techniques to tackle the insider problem, and to support and pre-compute possible audit results. By separating the actual threat and attack from reachability, we can easily put existing models and formalisms on top of our model.

We are currently working on extensions of this model to *malicious* insiders, who try to obtain keys as part of their actions in a system, and to further extend both the system model and the precision and granularity of our analyses.

References

1. M. Bishop. The Insider Problem Revisited. In *Proc. of New Security Paradigms Workshop 2005*, Lake Arrowhead, CA, USA, Sept. 2005. ACM Press.

2. V. L. Caruso. Outsourcing information technology and the insider threat. Master's thesis, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2003.
3. CERT/US Secret Service. Insider threat study: Illicit cyber activity in the banking and finance sector, August 2004. available at `www.cert.org/archive/pdf/bankfin040820.pdf`.
4. R. Chinchani, A. Iyer, H. Q. Ngo, and S. Upadhyaya. Towards a theory of insider threat assessment. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 108–117. IEEE Computer Society, 2005.
5. M. Dacier and Y. Deswarte. Privilege graph: an extension to the typed access matrix model. In *Proceedings of the European Symposium On Research In Computer Security*, 1994.
6. D. Gollmann. Insider Fraud. In B. Christianson, B. Crispo, W. S. Harbinson, and M. Roe, editors, *Proc. of the 6th International Workshop on Security Protocols*, volume 1550 of *Lecture Notes in Computer Science*, pages 213–219, Cambridge, UK, Apr. 1998. Springer Verlag.
7. J. Gorski and A. Wardzinski. Formalising fault trees. In F. Redmill and T. Anderson, editors, *Achievement and Assurance of Safety: Proceedings of the 3rd Safety-critical Systems Symposium*, pages 311–328, Brighton, 1995. Springer.
8. R. R. Hansen, C. W. Probst, and F. Nielson. Sandboxing in myKlaim. In *The First International Conference on Availability, Reliability and Security, ARES'06*, Vienna, Austria, Apr. 2006. IEEE Computer Society.
9. G. Morrisset. Personal communication, April 2006.
10. R. D. Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998.
11. H. R. Nielson and F. Nielson. Flow Logic: a multi-paradigmatic approach to static analysis. In *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer Verlag, 2002.
12. R. Ortalo, Y. Deswarte, and M. Kaâniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering*, 25(5):633–650, 1999.
13. J. Patzakis. New incident response best practices: Patch and proceed is no longer acceptable incident response procedure. White Paper, Guidance Software, Pasadena, CA, September 2003.
14. R. H. A. Richard C. Brackney, editor. *Understanding the Insider Threat*. RAND Corporation, Santa Monica, CA, U.S.A., March 2005.
15. E. D. Shaw, K. G. Ruby, and J. M. Post. The insider threat to information systems. Security Awareness Bulletin No. 2-98, Department of Defense Security Institute, September 1998.
16. L. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. June 12 2001.