available at www.sciencedirect.com

ScienceDirect

www.compseconline.com/publications/prodinf.htm

Information Security Technical Report

An extensible analysable system model☆

Christian W. Probst[a,*], René Rydhof Hansen[b]

[a]Technical University of Denmark, Denmark
[b]Aalborg University, Aalborg, Denmark

ABSTRACT

Analysing real-world systems for vulnerabilities with respect to security and safety threats is a difficult undertaking, not least due to a lack of availability of formalisations for those systems. While both formalisations and analyses can be found for artificial systems such as software, this does not hold for real physical systems. Approaches such as threat modelling try to target the formalisation of the real-world domain, but still are far from the rigid techniques available in security research. Many currently available approaches to assurance of critical infrastructure security are based on (quite successful) ad-hoc techniques. We believe they can be significantly improved beyond the state-of-the-art by pairing them with static analyses techniques.

In this paper we present an approach to both formalising those real-world systems, as well as providing an underlying semantics, which allows for easy development of analyses for the abstracted systems. We briefly present one application of our approach, namely the analysis of systems for potential insider threats.

1. Introduction

Over the last years, protection of critical (information) infrastructure has gained considerable importance. As a result, many approaches have been developed that, often based on an analysis of previous attacks, try to determine whether a currently observed activity should be deemed an attack or not. This approach has proved to be very successful, especially as often the behaviour observed does not change too much, thereby allowing the recognition of attacks.

On the other hand, static analysis (Nielson et al., 1999) is completely independent of previously observed behaviour or attacks—it is concerned with identifying properties that hold for every single configuration of the analysed system. Being based on a formal system model, from an initial configuration all possible states of the system are computed.

We believe that many of the existing approaches to the protection of critical information infrastructure can benefit from being paired with static analysis techniques. As we have shown in previous work (Probst et al., 2006), static analysis techniques can be used to compute possible threats *before the fact*, allowing to either identify a need to increased protection, or a need for special alertness, either at certain locations or at certain actors.

This allows our techniques to be used in the complete lifetime of a system—from the design of systems, over the prediction of possibly precarious situations during operation, to the guidance of auditing after an attack.

The rest of the paper is structured as follows. The next section introduces our abstract system model, along with an example, and Section 2.3 shows possible extensions of this model. Section 3 shows how systems can be represented in files, which then can be analysed using the techniques presented in Section 4. We conclude the paper in Section 5 by giving an outlook on future work. We start, however, by briefly introducing the problem of insider threats.

☆ Part of this work has been supported by the EU research project #016004, *Software Engineering for Service-Oriented Overlay Computers*.
* *Corresponding author*.
E-mail addresses: probst@imm.dtu.dk (C.W. Probst), rrh@cs.aau.dk (R.R. Hansen).
1363-4127/$ – see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.istr.2008.10.012

1.1. *Countering insider threats*

As modern economies depend ever more on information technology systems, the information handled by these systems becomes a precious good in lockstep. The disruption of services or loss of data can cause increasingly severe damage, leading to an ever increased interest in the protection of both data and systems.

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 infrastructure. The problem is well recognised in the security community as well as in law-enforcement and intelligence communities, cf. (Bishop, 2005; Brackney and Anderson, 2005; Gollmann, 1998; Bishop et al., 2008). 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 (Brackney and Anderson, 2005).

The biggest problem is that typically insider threats occur in real-world systems with, *e.g.*, large office complexes, human actors, and physical objects such as folders, printouts, and keys. While many analysis techniques exist for verifying safety and security properties, they have been developed for application to rigorous formal models, which usually do not exist for real-world systems. Formal modelling and analysis, however, is increasingly important in a modern environment with widely distributed (physical and computer) systems, computing grids, and service-oriented architectures, where the line between the real and the virtual domain is more blurred than ever. Approaches such as threat modelling (Swiderski and Snyder, 2004) try to target the formalisation of the real-world domain, but still are far from the rigid techniques available in security research and formal methods.

Before presenting our techniques for using such formal methods for identifying insider threats, we first want to define, what we understand by an insider, as this recently has been the topic of intense discussions, for example (Bishop, 2005; Brackney and Anderson, 2005; Bishop et al., 2008). Often, insiders and outsiders are treated the same since they can cause the same damage once they have the same knowledge. However, they intrinsically remain different with respect to the organisation they damage—while the insider is trusted to perform certain actions, the outsider most certainly is not. In our work we therefore make a clear distinction between *insiders* and *outsiders*, where only the former can be source of insider threats. However, as a *consequence* of such a threat, an outsider may obtain knowledge that will enable him to cause damage comparable to that an insider may cause. It is noteworthy that the techniques presented in this work can be applied to both insiders and outsiders.

2. View of the world

In this section we introduce the kind of system we are addressing with our model, and thereafter introduce the actual model in Section 2.2. There is no real restriction as to which properties of systems can be modelled, such that our approach can be used in almost any setting with varying granularity.

2.1. *High-level overview*

The kind of systems we are interested in is characterised by certain basic properties. We assume that there are *locations* that are *connected* by some means, and that there are *actors* who can move around in the system, performing *actions* on *data* as they move around. Again, the notions of locations, actors, actions, and data are rather loose—locations can, *e.g.*, be physical locations such as rooms or offices, or virtual locations in computer systems. In the former case the actors would probably be persons, in the latter programs. The actions performed can either be related to moving around, *e.g.*, entering or leaving a room, or to operations on data, such as access, creation, etc.

Consider, for example, the insider problem as introduced in Section 1.1. Clearly we are mostly concerned with office buildings and real data such as folders, as well as computer networks and data that is available on these. Fig. 1 shows such a real-world system components typical for the kind of systems we are interested in—namely rooms that are connected, a computer network, access control, etc. All these are aspects that we want to represent in our model, as they influence the way actors can behave in the modelled systems. In the example, actors can, *e.g.*, walk around, pick up data from the printer or the waste basket, etc. The example shown in the figure models part of an environment with physical

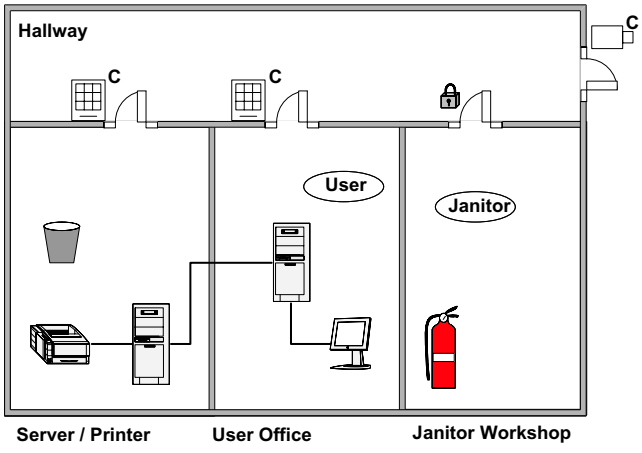

Fig. 1 – The example system used to illustrate our approach. The user can use the network connection to print some potentially confidential data in the server room. Depending on the configuration on the cipher locks, the janitor might or might not be able to pick up that printout. The building entrance is secured with a face recognition system.

locations (an entrance, a server room with a printer and a waste basket, 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 entrance, the server room, and the user office is restricted by a cipher lock, while the janitor workshop is accessible with a physical key. The actors in this system are a user and a janitor.

2.2. *An analysable model of the world*

We now define and discuss our system model in detail. The formal underpinnings of the modelling language will also be introduced and discussed to some extent, although the focus will be on the modelling aspects of the language. This includes specification of actions of interest, data items handled, and actors and their (partly) behaviour, if known.

The presented model is an extension of the one presented in Probst et al. (2006). Beside the model itself we also present how it can be extended with specialised constructs, and illustrate the extensibility with examples.

The abstraction is based on a system consisting of components. We distinguish between *location components*, such as offices and computers, *data components*, such as keys and actual data, and *mobile components*, such as processes and actors. Data can be associated with (stored at) locations and actors, and it can be secured by, *e.g.*, encryption, and locations can be secured by access-control mechanisms, *e.g.*, cipher locks. To support movements of dynamic components, locations can be connected by directed edges, which define freedoms of movements of actors.

The rest of this section introduces all of the system components and shows several possible extensions.

2.2.1. *Infrastructure*

We start with defining the notion of an *infrastructure*, which consists of a set of *locations* and *connections*. The infrastructure models the available connections in the modelled system, be it connections between rooms or computers, or be it possible to access one location from another one.

For the example shown in Fig. 1 we obtain the graph representation in Fig. 2. All rooms have become nodes in the graph, as have computers, the printer, and the waste basket. In general, all elements of a system where data can be located are modelled as nodes. Additionally, places at which some kind of access control is applied can be turned into nodes. In the example these are all the doors. The connections between nodes are generated based on the type of connection they allow in the real system. For example, there is a one-way connection from the node representing the hallway to the node representing the server room's cipher lock, since the user will have to pass the access control to get into the room. On the other hand, there is no check to get out of the room into the hallway, so there is a direct edge from the server room node to the hallway node.

Note the special node labelled "Outside", which represents parts of the system that are of no interest, in this case everything that is outside of the office complex. Collapsed nodes like this, that model potentially large areas of the system under inspection that are of no interest, allow the system modeller to concentrate on the essential parts to be modelled. If the analysis result turns out to be too imprecise, collapsed nodes can later be replaced by a more detailed model of the previously ignored part of the system. A system model can have several of these collapsed nodes, each representing entities that are not connected to each other. Choosing what to collapse is highly dependent on the modelled system and the analysis to be performed—in the case of the insider analysis our foremost interest is to find out whether data leaves the modelled system, therefore we are not interested in what lies outside of that system.

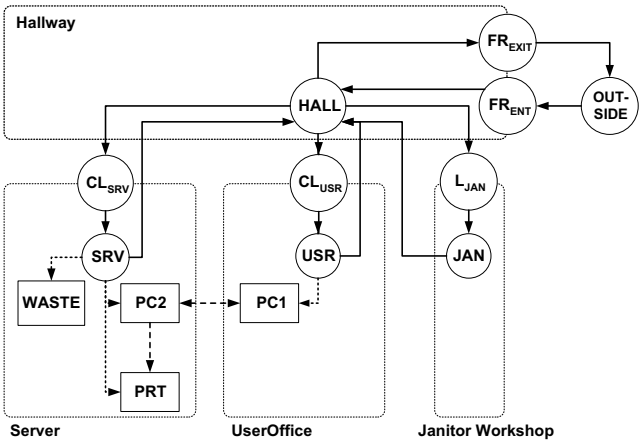

Fig. 2 – Abstraction for the example system from Fig. 1. The different kinds of arrows indicate how connections can be accessed. The solid lines, *e.g.*, are accessible by actors modelling persons, the dashed lines by processes executing on the network. The dotted lines are special in that they express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

As seen in Fig. 1, it is often natural to group several locations together, *e.g.*, based on the room they are located in. Groups of locations can provide interfaces that control access how actors can enter (some of the) locations in the group. In the example, the cipher locks prevent actors who do not have the correct code from entering some rooms, while the computers located in these rooms are accessible via the network.

2.2.2. *Actors*

Next, we define *actors*, which can move in the infrastructure by following edges between nodes, and *domains*, which allow to constrain the nodes that an actor can move in. Usually, actors can only move in a certain domain. In the example setting, actors would be the user, the janitor, or processes on the computers.

The user and the janitor can move in the locations representing rooms (physical domain), but they can only access, *e.g.*, the printer and the waste basket to take items out of them (object domain).

2.2.3. *Data*

Besides actors we are interested in the objects they work with, or *data* in general. Note that we use a rather loose definition of data—any set of items can be used to model data at

a convenient level of detail. Data items can be associated with both actors and locations, representing either what a user knows or possesses, *e.g.*, by carrying around, or what data is available at a certain location. Assigning such knowledge to actors or locations before performing an actual analysis allows to incorporate previously obtained insights into the analysis results—and furthermore allows to very easily test hypotheses about what could have happened if, *e.g.*, a certain actor had known a certain piece of information.

In the example, data could be used to model codes for the cipher locks or the key for the real lock, once we have extended the system model with access control in Section 2.3.1. Also printouts made from the work stations would be represented as data.

2.2.4. *Actions*

In order to model the behaviour of actors in a system, we will need to supply *actions* that can be performed by them. For each of these actions the system needs to specify how the action changes the locations of users, and the storage of data.

The actions we use are based on those available in the system described in Probst et al. (2006) and Gunnarsson (2007), partly because they allow to model the typical actions performed in real-world systems quite naturally. The actions allowed are input and output of data, evaluation of code at another location (starting a process on a computer), and moving to another location.

The effect of each action is currently described by the semantics of the underlying process calculus. We are working on integrating the effect specification into the system definition, in order to allow an even more modular specification, where the semantics can be freely substituted.

2.2.5. *System model definition*

A *system model* in our approach consists of all the components just introduced. Using locations, actors, data, and actions, it allows to capture the most important aspects of systems and insider threats—*who* the user is, *what* the user does and knows, and *where* the user does it. While very simple in nature, this model is both powerful enough to model real-world scenarios, and at the same time flexible enough to be easily extendable.

2.3. *System extensions*

In this section we briefly describe how to add extensions to the system model. The extensions covered are access control, encryption and decryption of data, which will be very similar to access control on locations, and a logging component, allowing to create log files, which then can be used in analysing the system.

2.3.1. *Access control*

To model systems with access control, we need to model how actors can obtain the right to access locations, and how access to a location can be granted or denied. We associate actors with a set of *capabilities*, and locations 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.

To ease presentation we use data items as keys and capabilities, and checker functions to test whether a given capability matches a restriction or not. In the example setting we could interpret the face of the actors as capability to enter the entrance of the building (based on face recognition), and the cipher keys as capabilities to enter the server room and the user office. The associated checkers would implement the test whether an actor with a given face is allowed to enter the building, or whether a cipher code matches the ones stored in the lock. In the semantics, access control and the related checkers are realised by reference monitors similar to Gunnarsson (2007); Hansen et al. (2006) and Probst et al. (2006) that check whether a certain action is allowed before executing it.

Fig. 3 shows the graphical representation of access control added to the example system (for the time being, ignore the overlined characters). Each of the boxes specifies how access is granted to certain actions at the location. For example, knowledge of the data item C_U allows both access to the server room as well as the user office, as specified by C_U:**m**, while C_J only allows access to the server room. The lines J:**m** and U:**m** at the entrance refer to the identity of the actor that wants to perform an action, and are here used to represent the face recognition.

2.3.2. *Encryption and decryption*

A convenient extension of the data model described in the previous section is to model encryption and decryption of data. We use an approach similar to the one for access control, that is we annotate data with the key it has been encrypted with, and require knowledge of the matching key for decryption. This can be seen as requirements (of the data for being decrypted) and capabilities (of the user for being able to decrypt). A data item can be encrypted with a set of keys—an empty set of keys represents unencrypted data.

It is noteworthy that this small change supports both asymmetric as well as symmetric encryption schemes, as the checker for the capabilities and restrictions ensures that the keys used for encryption and decryption match each other.

2.3.3. *Logging*

Another important component in a system is the ability to log (some of the) performed actions. Again, adding such a component to our system model is straightforward and requires only minimal changes to the system—the actual logging is performed as part of the semantics of the underlying process calculus. The additions to the system are a global clock and a logging component. The logging component maps a log entry to the reason why an action was allowed or denied—that is a certain key, the actor's identity, or the location from which the request came, as well as the locations from where to where the action was performed.

In the access-control specifications we distinguish logged and unlogged restrictions, and mark logged ones by using overlined characters (see Fig. 3). From the view of the system definition there is no difference between the versions with and without logging—all we assume is that there is a subset of restrictions that are logged as opposed to a subset of restrictions that are not logged. The difference will only occur in the underlying semantics.

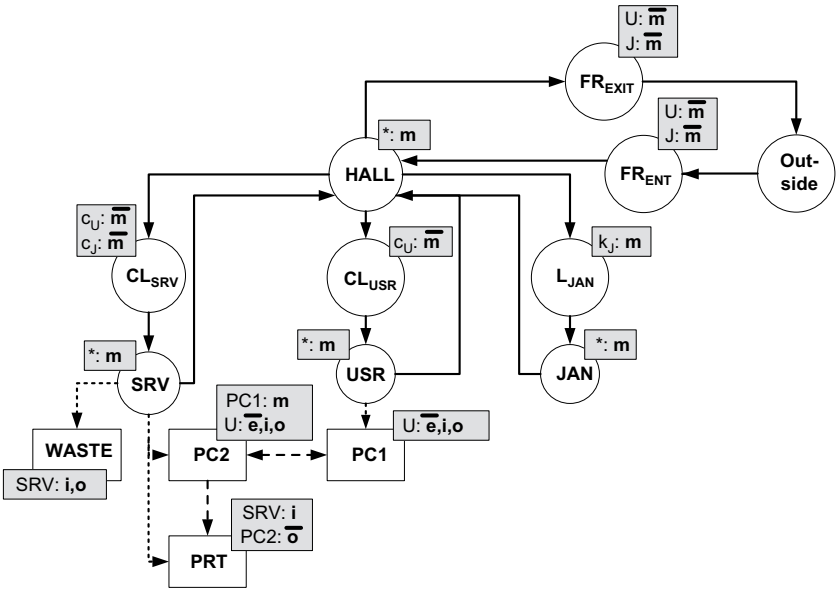

Fig. 3 – The abstracted example system from Fig. 2, extended with policy annotations. There are two actors, janitor J and user U, who, *e.g.*, have different access rights to the user office and the server room. Note the difference between accessing the user office or the server room with a cipher lock (logged) as opposed to the janitor workshop with a key (not logged).

2.4. *Formal semantics of system models*

The system model defined above has been designed to be easy and intuitive for use in the system modelling process. It furthermore lends itself very well to graph-based analysis techniques (see Section 4 for an example). However, in order to bring a wider variety of advanced analysis techniques and methodologies to bear, the system model needs a formal underpinning.

In Probst et al. (2006) we show how the semantics of a system model can be formalised using a *process algebra*. Process algebras have been widely used to model and study problems in the concurrency and distributed systems communities. This work has led to the development of numerous automated tools and advanced techniques for analysis and verification of system properties. By formalising the semantics of system models using process algebras we enable the use of these tools and techniques for forensic analysis of our system models.

The process algebra used in Probst et al. (2006) is called *acKlaim* and belongs to the Klaim family of process calculi originally developed to study the tuple-space paradigm (Nicola et al., 1998). The acKlaim calculus is a variation of the μKlaim calculus enhanced with access-control primitives and equipped with a *reference monitor semantics*, inspired by Hansen et al. (2006), that ensures compliance with a system's access-control policy. In addition to providing a convenient and well-understood formal framework with built-in support for access control, it has a well-established and mature history showing numerous successful applications of formal methods and techniques to analysis and verification of Klaim systems (Nicola et al., 1998; Nicola et al., 2000). This provides analysts with a proven and well-tested toolbox for analysing system models. In addition it forms a solid foundation for exploring novel approaches and integrating new technologies, thereby enabling forensic analysts to adapt the analysis platform to emerging threats and the ever-changing attack landscape.

The reference monitor semantics mentioned above ensures that the semantics only performs actions that are specifically allowed, based on a number of factors, such as: the type of action to be performed, the identity of the actor wishing to perform the action, the data in possession of the actor, and/or the locations involved in the action. This constitutes an implementation of the checkers defined in the previous section, and is also the place where logging is added to the system.

3. A modelling language

Being based on a collection of mathematical definitions, the abstract system specification described so far is not well suited for implementation. In this section we therefore define a language for specifying system models as text files, which then can be used as input for analyses. Although an abstract specification will be mapped to an acKlaim program, the syntax of the language should be as close to the abstract specification as possible. However, the user should not have to know anything about the underlying theory in order to use the analyses.

The syntax of the specification language is given in Fig. 4 and Fig. 5. Like the specification from Section 2.2, a system is composed of four major categories: locations, connections, actors, and data. Locations are represented by a location name along with a list of restrictions that the location makes on actions performed on it. Each restriction is a name specifying a location, an actor, a piece of data, or ★ representing a wild card, and a list of actions that the given name is allowed to perform at the location. Each location also specifies the

```
Spec ::= locations: { Locations }
         connections: { Connections }
         actors: { Actors }
         data: { Data }
Locations ::= Location *                                          list of locations
Location ::= LocationName { LocationPolicies } ( DomainName ) ;  a location in a domain
LocationPolicies ::= Policy *                                     list of policies
Policy ::= Names : AllowedActions ;
         | ⋆ : AllowedActions ;
AllowedActions ::= LocationAction *                               list of allowed action
LocationAction ::= i | ilog                                       input
         | o | olog                                               output
         | e | elog                                               spawn a process
         | m | mlog                                               move
```

Fig. 4 – Syntax for specifying locations and associated policies. Elements of lists are separated with commas. Note that for the list of allowed actions only one version (logged or unlogged) of each action may occur.

domain it is member of. The domain is simply a name and must of course not conflict with names used for other purposes. The list of restrictions for a location may be empty, meaning that no restrictions are imposed on the access of that location. To model that no access is allowed by anyone, the list of access modes should be left empty as in safe {⋆:}.

Connections are specified with a right-pointing arrow from a location A to another location B, meaning that there is an edge from A to B. Both end points in a connection must be defined for the connection to be well-formed.

Actors are represented by a list of pairs of actor names and the name of the location(s) the actor is located at initially. Note that, in the case of uncertainty, an actor may be placed at several locations. The set of actor names must be disjoint from the set of location names for the specification to be well-formed. The location at which an actor is located must be defined in the list of locations.

Data is specified as a list of data elements annotated with access restrictions and information on where they are located (either at an actor or at a location). For ease of presentation, the structure of data is one-dimensional, namely a single string. This can easily be extended to a more complex tuple structure with nested tuples and so on. It should be noted that such a change does not require changes to the techniques presented here. If the list of restrictions is empty, the datum is assumed to be public, and if the list of access modes is empty for the ⋆ name, the datum cannot be accessed by any actor in the system. Access restriction on data may be decryption and read/write restrictions, modelled as input and output as for locations. Any actor is free to pick up or read data (as long as he has access to it), but to get the information that encrypted data holds he needs to have the necessary key to be able to decrypt it.

The text in Fig. 6 shows the representation of the example system from Fig. 3. Beyond the system graph shown there, the

```
Connections ::= Connection *                                  list of connections
Connection ::= LocationName -> LocationNames ;
Actors ::= Actor *                                            list of actors
Actor ::= ActorName @ LocationNames                           actor and start location(s)
Data ::= Datum *                                              list of data items
Datum ::= DataName { DataPolicies } @ LocationName ;          data at a location
        | DataName { DataPolicies } @ ActorName ;             data at an actor
DataPolicies ::= DataPolicy *                                 list of data policies
DataPolicy ::= Names : DataActions ?
        | ⋆ : DataActions ?
DataActions ::= d | dlog                                      decrypt
        | i | ilog                                            read file
        | o | olog                                            write file
```

Fig. 5 – Syntax for specifying connections, actors, and data. The same restrictions for lists apply as in Fig. 4. Specification of an action in a *DataPolicy* is optional.

```
locations: {
  outside  { } (building);
  entry    { U: mlog; J: mlog; } (building);
  exit     { U: mlog; J: mlog; } (building);
  hall     { ⋆: m; } (building);
  lock_jan { key_jan: m; } (building);
  jan      { ⋆: m; } (building);
  lock_usr { code_U: mlog; } (building);
  user     { ⋆: m; } (building);
  pc1      { U: elog, i, o; } (virtual);
  lock_srv { code_U: mlog; code_J: mlog }
             (building);
  server   { ⋆: m; } (building);
  pc2      { pc1: m; U: elog, i, o; }
             (virtual);
  printer  { srv: i; pc2: olog; } (device);
  waste    { srv: i,o;} (object);
}
```

```
connections {
  outside -> entry;
  entry -> hall; exit -> outside;
  hall -> lock_jan, lock_usr, lock_srv,
                exit;
  lock_jan -> jan; jan -> hall;
  lock_usr -> usr; usr -> hall; usr -> pc1;
  pc1 -> pc2; pc2 -> pc1
  lock_srv -> srv;
  srv -> hall, waste, pc2, printer;
}

actors {
  U @ outside;
  J @ outside;
}

data {
  code_U { } @ U;
  code_J { } @ J; key_jan { } @ J;
  secret_file { U: r } @ pc1;
}
```

Fig. 6 – Textual representation of the example system from Fig. 3, including some data known at actors or stored at locations. For experiments, the locations of actors and data can easily be changed.

textual representation also includes the data available at actors or locations, as well as the locations where actors are located initially. In order to run through different scenarios, these locations, as well as the data available, can easily be changed. If, for example, during an investigation there was some doubt, where the janitor was located, the input could be changed to J @ outside, server if he might have been at the server room.

4. System model analysis

In this section we present two analysis techniques related to the model and language described in the previous sections. The goal of this section is to introduce the reader to some of the analytical tools and insights needed to get started. Both analyses are graph-based and work on the system model defined above.

The first analysis, called *Conditional Reachability Analysis*, is designed to determine, 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. In the insider-threat scenario introduced above, this allows to determine, which locations an insider can reach and which data he can potentially access. 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, called *Log-trace Reachability Analysis*, takes a log file as input and based on this determines which actor has been where, performing which actions and accessing which data. In the insider-threat scenario this analysis allows to determine where actors might have been based on the observed actions. In an after-the-fact analysis, this analysis can also be used to add observations or investigation results to the log file in order to evaluate their impact.

The rest of this section is structured as follows. We first give a high-level introduction to the two analyses, followed by a more detailed presentation of their technical details in Section 4.2. After this, we evaluate both analyses in Section 4.3 by applying them to the example system. Interested readers can find the technical details in Section 4.2.

4.1. *Analysis overview*

When dealing with insiders and the threat they potentially might pose, we deem two scenarios especially important—on the one hand, we would like to know *before* an attack, what capabilities certain actors have in the system based on what they know (and, as a result of this, where they can get). On the other hand, once an attack has happened, we would like to be able to identify, what has happened in the system before, under, and after the attack. Before presenting the technical details in the next section (which safely can be skipped), we give a more high-level overview, how these two analyses work on our system model. It should be noted that these analyses only are examples of what the system can be used for. We are currently working on control-flow analyses (Nielson et al., 1999), which however are beyond the scope of the work presented here. An example can be found in Probst et al. (2006).

4.1.1. *Before the fact*

When designing a system, especially an access-control system, it rapidly becomes unclear, which parts of the system are accessible by which users. In systems combining networks with real buildings, the distinction between reachable and unreachable becomes even more blurry.

In this scenario our first analysis may be applied. Given a representation of the system under development (Section 2.2), and an extension with access restrictions (Section 2.3.1), this analysis allows to identify which places a user may reach, based on who he is, what he knows, and where he is located. This analysis has two immediate applications—it allows to identify possible shortcomings in an access-control system, and it allows to decide whom to use in order to reach a certain location or retrieve a certain piece of data. While the first application probably is obvious, the second might not be. Here the idea is to use the analysis to find out which users are able to reach a certain location, based on their identity and knowledge (the who and what above). From all these users, one then can chose the user who lives up to certain expectations. These might be, for example, fewest access rights (meaning that the potential collateral damage is minimised), or above a certain rank in the hierarchy (hopefully meaning that this user can be trusted more than users below him in the hierarchy). Beyond these are uncounted possibilities to use and interpret the results of this first analysis.

The analysis that provides system designers with this knowledge is the conditional reachability analysis (CRA, Section 4.2.2). It receives a system model such as the one from Fig. 6 as input, and simulates for all users all actions they may perform. In this process, a user may be located at several positions simultaneously, thus representing uncertainty as to where the user is located exactly. For each user the analysis traces all possible ways the user might take through the system. While this may occur pessimistic given that the user eventually will only perform one sequence of actions, it is at the same time conservative in computing a super set of what will happen in real life. This is a necessary property for each analysis whose results can be applied in a meaningful way—if the analysis computes a certain result it must somehow be possible that this result occurs (Nielson et al., 1999). Even though the analysis thus computes many ways through the system that no actor ever will follow, the computed result still allows to see what can happen in the system based on access rights assigned to locations and keys assigned to actors—exactly what is needed for a before-the-fact analysis to support the system designer.

4.1.2. *After the fact*

Having designed a system, possibly with help of an analysis like the just described, one needs to prepare for a potential attack. Such a preparation can come in several forms—be it in form of logging of actions performed by users, be it in form of after-the-fact forensic analyses using the log data as input. Surprisingly, this after-the-fact analysis still seems to be applied frequently (Brackney and Anderson, 2005).

While in Utopian worlds complete surveillance often is assumed to be acceptable (and accepted), this is certainly not the case for our societies. Privacy concerns often limit the amount of data that may be logged[1], thus also limiting how useful the logged information is. Even worse, it often is more interesting what might have happened (unnoticed) in between two log entries, than what actually has been logged. In this situation our second analysis may help. Like the before-the-fact analysis it receives a system model as input (Section 2.2), this time with the logging extension (Section 2.3.3), and a stream of logged events, for example recovered from some kind of logging system. This could either be the dump of logging units in the system, it could, however, also be observations made as part of an investigation, or a mix of both sources. Based on these, the analysis explores what an actor might have done in between two log entries. The more fine-grained the logging system is, the more precise the result of this analysis will be, but the more coarse-grained the logging system is, the more beneficial is our analysis. This is because the set of actions possibly performed between two log entries is getting bigger and bigger the further the two logged actions are apart. Consequently, it becomes harder and harder to keep a clear view of what might have happened in between.

The analysis that provides investigators with this knowledge is the log-trace reachability analysis (LTRA, Section 4.2.3). It receives a system model such as the one from Fig. 6 and a set of logged events as input, and simulates for all users all actions they may perform such that the set of logged events is generated. The overall operation of this analysis is very similar to the analysis described before—again, the analysis traces for all users all possible ways that they might take through the system. However, in this analysis the set of all paths is restricted by the requirement that the actions performed must match the logged events. This restriction results exactly in what is needed for the after-the-fact analysis—a tool that matches logged actions against possible actions, thus identifying locations that an actor might have reached and data items an actor might have accessed unnoticed.

4.2. *Technical details*

This section introduces some of the technical details underlying our analyses, including pseudo-code algorithms for computing the analysis results. Before going in more details with respect to the two analyses, we first discuss equivalent locations, an important issue for both analyses presented here.

4.2.1. *Equivalent locations and actions*

A notion that we will use several times in the following discussion is that of *equivalent locations and actions*. By this we mean locations and actions that from the viewpoint of an observer, in our case the analysis, cannot be distinguished. As a result, if the analysis finds out that an actor can be in a location ℓ, then he might just as well be in *any* equivalent location, or might have performed any actions in between.

This notion of equivalence serves two different purposes. In the case of the conditional reachability analysis (Section 4.2.2) we use it to speed up the analysis—since the actor could reach all equivalent locations anyway, it is easier to just compute the transitive, reflexive hull of the current location and assume the actor is at any of these or has performed any actions possible in between.

In the case of the log-trace reachability analysis, equivalency of locations and actions is defined based on whether or

[1] Depending on the kind of institution applying the logging these restrictions may be abandoned by individuals by signing according contracts. This may be deemed against public policy and inoperative in many states.

```
1: equivalent()
2: /* perform (log-)equivalent actions */
3: changed = true
4: while changed do
5:   changed = false
6:   for all actors n do
7:     for all locations l that n might be located at do
8:       for all locations l' reachable from l in one step do
9:         simulate all actions that n can perform on l' (without causing a log entry
           in case of LTRA)
10:        for each action set changed if n at location l learns a new data item
11:      end for
12:    end for
13:  end for
14: end while
```

Fig. 7 – For each actor in the system we check for all locations he can be located at whether he can perform any actions. All these actions are assumed to have been performed. In the case of the log-trace reachability analysis, only actions that would not cause a log entry are considered.

not reaching a location from another one or performing an action causes a log entry. Here, two locations and/or actions are deemed indistinguishable from the viewpoint of the analysis if the actor does not cause a log entry. Log-equivalency is needed to find out what might have happened between two log entries.

The pseudo code in Fig. 7 shows the realisation of log-equivalency in the log-trace reachability analysis. It simply visits all locations where a user might be, and computes the effect of every unlogged action that the user is allowed to perform at that location. This computation is repeated until no further changes to the graph occur. The implementation of regular equivalency is quite similar, the only difference being that there is no restriction as to causing a log entry.

4.2.2. *Conditional reachability analysis*

As mentioned before, the conditional reachability analysis is equivalent to a before-the-fact analysis, where a system designer might want to determine whether a given system lives up to a set of access-control restrictions. To do so, the analysis assumes the worst case—that is, what ever can happen, will happen. This is especially crucial with respect to data exchange, which in our case means that keys or secret data might be handed over between actors in the system.

Like the log-trace reachability analysis, which will be covered in the next section, this analysis is graph based. It starts from a system specification as presented before, constructs a graph from it, and for each actor simulates all possible actions that are allowed by the system. While the LTRA will restrict possible paths through the system with the set of logged actions that have been observed, the conditional reachability analysis explores the whole graph unconstrained, performing every action possible. In order to avoid non-termination it keeps track of which actor with which knowledge has been at which location—thus, re-analysing already seen scenarios can be avoided.

The algorithm for CRA is given in Fig. 8. Essentially it only sets up the analysis by initialising all data structures, followed by a single call to equivalent, which performs the simulation of all possible actions and iterates until no further changes occur.

4.2.3. *Log-trace reachability analysis*

In Fig. 9 we present a graph-based algorithm for evaluating, what effect sequences of logged actions might have had, by evaluating all sequences of actions on the system representation. Note that the algorithm is intended to demonstrate the principles underlying the solution rather than being an optimal implementation.

The algorithm works on the sequence of logged actions. To trace the potential actions of actors, it traces where actors might be located, which data an actor *n* at a location *l* knows, and which data is stored at which location. Initially, all actors are assumed to be outside the system (location outside in the example) and to know an initial key set, which may be empty. Also locations are initialized with the potentially empty initial data set.

Following the initialization, the algorithm consumes all entries in the log sequence. During each iteration it first simulates all log-equivalent actions that might be executed by actors at their current locations. This simulation is repeated

```
1: ConditionalReachability
2: Input : system specification
3: /* initialization */
4: place all actors at their initial location
5: initialize all actors and locations with initial key set
6: /* compute all actions that can be performed */
7: equivalent()
```

Fig. 8 – Algorithm for computing which places an actor may reach in the system, based on the actor's initial location and knowledge. Initially, all actors and locations are initialized with the data they are assumed to know beforehand, and actors are located at their possible locations. Thereafter, the algorithm only needs to call the function equivalent (Fig. 7), which computes and simulates iteratively all actions that can be performed, until a fixpoint is reached. At the end, for each actor we know all locations, and for each pair of actor and location we know the knowledge at this point.

```
1: Log – TraceReachability
2: Input : system specification and log sequence
3: /* initialization */
4: place all actors at their initial location
5: initialize all actors and locations with initial key set
6: /* iterate over log sequence */
7: while log sequence not empty do
8:   /* perform log-equivalent actions */
9:   equivalent()
10:  /* consume the next logged action */
11:  pick next (reason, from, to, action) from log sequence
12:  if reason is an actor then
13:    from is the exact location of the actor reason
14:    remove that actor from all other locations
15:    the only possible actor is reason
16:  else if reason is a key then
17:    possible actors are all actors who might be at from and know the key reason
18:    if only one actor at from knows the key reason then
19:      remove that actor from all other locations
20:    end if
21:  else if reason is a location then
22:    potential actors are all actors who might be at from
23:    if only one actor is located at from then
24:      remove that actor from all other locations
25:    end if
26:  end if
27:  for all potential actors n do
28:    simulate effect of n performing action action
29:  end for
30: end while
31: /* perform log-equivalent actions */
32: equivalent()
```

Fig. 9 – Algorithm for evaluating the possible effect of all sequences of actions that can cause the logged events. For each logged event the algorithm performs all actions that could go unnoticed (line 9), and identifies the set of actors that possibly can have caused it (lines 12–26). If only one actor can have performed a logged action, we know exactly where this actor is located, and it consequently is removed from all other locations (lines 14, 18–20, 23–25). Finally, the effect of the logged event is simulated.

until no further changes occur. After having simulated all log-equivalent actions in the current state, the next logged action is consumed. Before simulating this action, the algorithm first checks whether there is exactly one actor that can have caused the log entry, in which case the data structures are updated accordingly.

Finally, at the end of each iteration the logged action is simulated for all actors that may have caused the action to happen. After doing so, the iteration starts over with the next logged action, until all actions have been consumed. The algorithm then repeats the simulation of all log-equivalent actions.

4.3. *An example*

Now we apply the two analyses just presented to the example system shown in Figs. 3 and 6. We start with the conditional reachability analysis (Section 4.2.2). As shown in the textual representation, we expect the user and the janitor to be located outside the system, knowing the codes and/or keys to their office and the server room. Table 1 shows the result of applying CRA to the example. The table contains analysis results for two different cases—first we analyse the cases where the user and the janitor each are alone in the system. As one would expect, the analysis finds out that each of them can access the rooms they have the keys to, and that the user can obtain the secret file stored on pc1, either by printing it on the printer in the server room, or by obtaining it directly from pc1. The file is only readable by U, therefore the janitor cannot obtain it. In analysing the second case we assume the user and the janitor to be acting in the system simultaneously. In this case, the janitor is able to get hold of the secret file, namely if the user prints it in the server room. It should be noted that this second case is a coarse approximation of what really might happen, as it does not contain any information about time—nevertheless, the threat exists.

Table 1 – Result of the conditional reachability analysis for the example system from Figs. 2 and 6. As to be expected the user can obtain the secret file (either directly from pc1 or by printing it and picking it up from the printer). The janitor, on the other hand, cannot access the file. When both actors are analysed simultaneously, then the janitor can access secret because the user might print it, and the janitor has access to the server room.

actor	location	data
analysis result		
U	outside, entry, exit, hall, $\text{lock}_{\text{user}}$, user, lock_{srv}, server	code_{U}, secret
	lock_{jan}, jan	∅
J	outside, entry, exit, hall, lock_{jan}, jan, lock_{srv}, server	code_{J}, key_{jan}
	$\text{lock}_{\text{user}}$, user	∅
simultaneous analysis result		
U	outside, entry, exit, hall, $\text{lock}_{\text{user}}$, user, lock_{srv}, server	code_{U}, secret
	lock_{jan}, jan	∅
J	outside, entry, exit, hall, lock_{jan}, jan, lock_{srv}, server	code_{J}, key_{jan}, secret
	$\text{lock}_{\text{user}}$, user	∅

The ordering relation based on time is taken into account in the log-trace reachability analysis (Section 4.2.3), which on top of the system description also gets a string of logged actions as input. There are two interesting cases with respect to the two actors we are investigating. Considering the file secret once it has been printed, it is of interest whether the janitor has been in and left the server room before the file is printed, or not. If not, then there is a risk of the janitor picking up the secret file from the printer—or even only reading it and leaving it in place. For performing LTRA we assume two different log sequences. The first one is generated by the user entering the system, going to his office, logging onto the system, printing the file secret, going to the server room, picking up the printout, and leaving the system again, followed by the janitor coming, going to the server room, and leaving again:

$(1, \text{U}, \text{entry}, \mathbf{m}), (5, \text{code}_{\text{U}}, \text{lock}_{\text{user}}, \mathbf{m}), (8, \text{U}, \text{pc1}, \mathbf{e}),$
$(10, \text{pc2}, \text{printer}, \mathbf{o}), (23, \text{code}_{\text{U}}, \text{lock}_{\text{srv}}, \mathbf{m}), (32, \text{U}, \text{exit}, \mathbf{m}),$
$(40, \text{J}, \text{entry}, \mathbf{m}), (45, \text{J}, \text{lock}_{\text{srv}}, \mathbf{m}), (58, \text{J}, \text{exit}, \mathbf{m})$

Note that the log sequence does not mention the file to be printed, but since it is stored on pc1 the analysis identifies it as potentially printed. The second log sequence describes a case where the janitor leaves the server room before the file is printed:

$(1, \text{J}, \text{entry}, \mathbf{m}), (11, \text{U}, \text{entry}, \mathbf{m}), (15, \text{J}, \text{lock}_{\text{srv}}, \mathbf{m}),$
$(17, \text{code}_{\text{U}}, \text{lock}_{\text{user}}, \mathbf{m}), (20, \text{J}, \text{exit}, \mathbf{m}), (22, \text{U}, \text{pc1}, \mathbf{e}),$
$(25, \text{pc2}, \text{printer}, \mathbf{o}), (26, \text{code}_{\text{U}}, \text{lock}_{\text{srv}}, \mathbf{m}), (32, \text{U}, \text{exit}, \mathbf{m})$

The result for these two sequences is shown in Table 2. It shows that for the first sequence the janitor may obtain the secret document—this is because we can not guarantee that the user picked up the printout, even though he was in the server room at time 23. This is because we cannot *observe* the picking up. For the second log sequence, we know that the janitor does not enter the server room after the file has been printed, and the analysis result confirms this.

Table 2 – Result of the log-trace reachability analysis for the example system from Figs. 2 and 6.

actor	location	data
log sequence 1		
U	outside, entry, exit, hall, $\text{lock}_{\text{user}}$, user, lock_{srv}, server	code_{U}, secret
	lock_{jan}, jan	∅
J	outside, entry, exit, hall, lock_{jan}, user, lock_{srv}, server	code_{J}, key_{jan}, secret
	$\text{lock}_{\text{user}}$, user	∅
log sequence 2		
U	outside, entry, exit, hall, $\text{lock}_{\text{user}}$, user, lock_{srv}, server	code_{U}, secret
	lock_{jan}, jan	∅
J	outside, entry, exit, hall, lock_{jan}, user, lock_{srv}, server	code_{J}, key_{jan}
	$\text{lock}_{\text{user}}$, user	∅

5. Conclusion

We have presented an extensible, analysable system model for real-world systems. While the system model originated in a desire to analyse and prevent insider attacks, the model is sufficiently general, and easily extendable as shown, that it can be used in many other application areas. In addition to the abstract system model we have presented a modelling language for representing and developing concrete models, and we have shown that the underlying model can easily be extended with domain specific concepts and notions such as access control, cryptography, and logging.

In earlier work a formal semantics for the abstract system model, in the form of a process algebra, has been specified and used to further extend the analyst's toolbox with methods and techniques from the programming language and program analysis communities. This is in contrast to many current approaches that often lack this formal underpinning. We believe that a formal semantics is absolutely essential for future development, both in order to better understand the underlying mechanisms, as well as for enabling a more formal and rigorous approach to dealing with insider threats. Even more so with the growing popularity and subsequent deployment of distributed and decentralised systems, as well as notions such as ''cloud computing'', grid computing, and Software as a Service (SaaS). As the protection of these (often mission-critical) information infrastructures has gained considerable importance in the last years, many approaches have been developed, which often are based on an analysis of previous attacks. While these approaches have been very successful, we believe that they can benefit from being paired with static analysis techniques as shown in previous work (Probst et al., 2006). Using these techniques, the model presented in this paper allows to either identify a need to increased protection, or a need for special alertness, either at certain locations or at certain actors.

A specific advantage of the flexibility of our approach is that it can be used throughout the entire life-cycle of a system—from the design, over the prediction of possibly precarious situation during operation, to the guidance of auditing after an attack. Furthermore, due to its foundation on

static analysis, it enables both combing sub-models, *e.g.*, of different parts of an investigations, to a bigger model, as well as adapting the granularity (and as a result the speed) of the analysis. We believe that these are properties essential for enabling development and analysis of large, real-world scenarios.

REFERENCES

Bishop M. The insider problem revisited. In: Proc. of new security paradigms workshop 2005. Lake Arrowhead, CA, USA: ACM Press; 2005.

Bishop M, Gollmann D, Hunker J, Probst CW. Dagstuhl seminar ''countering insider threats'', http://www.dagstuhl.de/08302; 2008. last visited [accessed 12.08.08].

Brackney RC, Anderson RH, editors. Understanding the insider threat. Santa Monica, CA, U.S.A.: RAND Corporation; 2005.

Gollmann D. Insider fraud. In: Christianson B, Crispo B, Harbinson WS, Roe M, editors. Proc. of the 6th international workshop on security protocols, vol. 1550 of lecture notes in computer science. Cambridge, UK: Springer Verlag; 1998.

Gunnarsson D. Static analysis of the insider problem, Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, supervised by Christian W. Probst, IMM, DTU; 2007.

Hansen RR, Probst CW, Nielson F. Sandboxing in myKlaim. In: The first international conference on availability, reliability and security, ARES'06. Vienna, Austria: IEEE Computer Society; 2006.

Nicola RD, Ferrari G, Pugliese R. KLAIM: a kernel language for agents interaction and mobility. IEEE Transactions on Software Engineering 1998;24(5):315–30.

Nicola RD, Ferrari G, Pugliese R, Venneri B. Types for access control. Theoretical Comput Sci 2000;240(1):215–54.

Nielson F, Nielson HR, Hankin CL. Principles of program analysis. Springer-Verlag; 1999.

Probst CW, Hansen RR, Nielson F. Where can an insider attack? In: Workshop on formal aspects in security and trust (FAST 2006); 2006.

Swiderski F, Snyder W. Threat modeling. Microsoft Press; 2004.

Christian W. Probst is Associate Professor in the Language-based Technology section at the Technical University of Denmark. He works on programming languages and modelling, analysis, and realization of systems, especially under security aspects.

Rene Rydhof Hansen is Assitant Professor at Aalborg University, Denmark. He works in the area of static analysis for safe and secure systems.