Static Analysis of the Insider Problem

Dagur Gunnarsson

Kongens Lyngby 2007
IMM-MSC-2007-14

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-MSC: ISSN 0909-3192

Summary

Every organization or company relies on data in one form or another both digital data and physical data. One of the main challenges companies and organizations face is securing data and other valuable assets. For some organizations security is more important than others, e.g., a bank's most valuable asset is its data, transactions and other financial data. Defining security policies is a major task, enforcing security policies an even bigger one.

Security policies should be defined to protect data from malicious attackers from the outside world as well as from people that have inside knowledge of the inner workings of the organization. Methods have been developed to secure the IT-infrastructure from the outside world, but there is not much focus on securing data from the inside.

In this thesis we address the problem of analyzing insider threats or the "insider problem" by using static program analysis methods. We develop a framework for specifying real-world systems and develop methods for finding insider threats in these systems.

Preface

This thesis was prepared at the Informatics Mathematical Modelling department, the Technical University of Denmark in partial fulfillment of the requirements for acquiring an M.Sc. degree in engineering.

The thesis deals with the use of static analysis for analyzing models of real-world system for insider threats. In particular, we develop a tool capable of analyzing system specification for insider threats. The tool is implemented in F# and C#. The output of our tool can be used to find weaknesses in real-world systems before an attack occurs, and also to guide the investigation of an attack after an attack has happened.

Reykjavik, February 2007

Dagur Gunnarsson

Acknowledgements

It is extremely hard to return to school after several years working in the IT-industry. Even harder is to move to a foreign country. I would first of all thank my wife and son for supporting me through this part of my education and understanding how important it was for me to finish my education.

I have had many interesting courses at DTU and met many interesting and smart people. I have had a really good relationship with many people at IMM. I got to know Christian W. Probst when he taught the Program Analysis course on my second semester and I knew right away that I wanted to do a thesis on static analysis. I would like to thank Christian for supervising my thesis and his excellent supervision through out the period. He always found time to listen to my thoughts and explaining complicated matters.

I would also like to thank Henrik Pilegaard and Terkel Kristian Tolstrup for taking time to discuss my thesis and give me good advice.

Contents

Summary **i**

Preface **iii**

Acknowledgements **v**

1 Introduction **1**

1.1 Our Work 1

1.2 Thesis Organization 2

2 Background **3**

2.1 Klaim 3

2.1.1 KlaimSyntax 4

2.2 Insider problem 5

2.3 Insider Work 6

2.3.1 The Abstract System Model 7

2.3.2 acKlaim Security Policies 8

2.3.3 acKlaim syntax 9

2.3.4 acKlaim semantics 11

2.3.5 Analysis 14

2.3.5.1 Reachability Analysis 15

2.3.5.2 Control-Flow analysis 16

2.4 Flow Logic 17

3 The Insider Framework 19

3.1 Framework Requirements 19

3.2 Insider Framework 20

3.3 Running Example 21

3.4 Abstract systems 22

3.5 Access Policies 24

3.6 System Specification Language 26

3.7 insCalc Syntax 29

3.8 Mapping Abstract Systems to insCalc 33

3.9 Semantics of insCalc 34

3.9.1 The Congruence Relation 34

3.9.2 The Reference Monitor 35

3.9.3 The Reduction Relation 37

3.10 The Analysis 42

3.10.1 Reachability analysis (Analysis0) 42

3.10.1.1 Algorithm for Analysis0 43
3.10.1.2 Running analysis0 44
3.10.2 Control Flow Analysis (Analysis1) 45
3.10.2.1 Flow Logic Specification 46
3.10.2.2 Language for Specifying Process Definitions . . . 48
3.10.2.3 Algorithm for Calculating the Analysis Estimate 48
3.10.3 Running Analysis1 51

4 Extensions to The Insider Framework 55
4.1 Abstract System with Logging 56
4.2 Access Modes 56
4.3 Syntax of The System Specification Language 57
4.4 Semantics 58
4.5 Analysis with Logging (Analysis2) 61
4.5.1 Example System 62
4.5.2 Algorithm for Analysis2 63
4.5.3 Running the algorithm 65

5 Program Design and Implementation 69
5.1 User Interface 69
5.1.1 Loading Systems 71
5.1.2 Analysis0 71
5.1.3 Analysis1 72

5.1.4 Analysis2 73
5.2 Program Source 75
5.2.1 Programming Languages 75
5.2.2 Programming Environment 75
5.2.3 Program Structure 75
5.3 Parsers and Lexers 78
5.4 Summary 79

6 Evaluation 81

6.1 Specification 1 81
6.1.1 Analysis0 82
6.1.2 Analysis1 83
6.1.3 Analysis2 85
6.2 Specification 2 87
6.2.1 Analysis0 88
6.2.2 Analysis1 89
6.2.3 Analysis2 91
6.3 Summary 91

7 Conclusion 93

7.1 Achievements 93
7.2 Limitations 94
7.3 Future Work 94

A Test Systems **97**

A.1 Specification1 97

A.1.1 Test1.spe 97

A.1.2 Test1.pde 98

A.1.3 Test1.log 98

A.2 Chain of Keys 98

A.2.1 Chain_Of_Keys.spe 98

A.2.2 Chain_Of_Keys.pde 99

A.2.3 Chain_Of_Keys.log 99

A.3 The Long Hall 99

A.3.1 The_Long_Hall.spe 99

A.3.2 The_Long_Hall.pde 100

A.3.3 The_Long_Hall.log 100

A.4 The PayCheck 101

A.4.1 PayCheck.spe 101

A.4.2 PayCheck.pde 101

A.4.3 PayCheck.log 102

Chapter 1

Introduction

Generally, information security focuses on protecting against attack from outside attackers, e.g. by use of fire walls. Little research focus has been on protecting IT-infrastructures from *insider attacks* and dealing with insider attacks. The main measure taken is still to audit log files after the attacks have been made to the infrastructure. IT-security is getting more attention and regularly you hear about breaches in IT-security that lead to financial loss or even worse a loss in the creditability of companies.

The insider problem is extremely hard to deal with because the attacker is a person with insider knowledge of the infrastructure, and knows how to utilize holes in IT-systems. It is also hard to come up with policies that prevent insider attacks as at some level employees are "trusted" to perform actions in a legitimum way.

1.1 Our Work

In this thesis we develop an implementable framework for doing insider analysis and implement a tool for analyzing systems for insider threats. We extend the work done by [16] and focused on creating an implementable framework with three different insider analyses. The first analysis finds an over approximation of the data and locations an actor in a system can reach if he uses every single option that he has in order to gain more access in the system. The second analysis finds an over-approximation of the data and locations that an actor can reach if he follows some sequence of actions in the system. The result will be a smaller set than the first one, as the actor is no longer "free" to do anything. The final analysis is a more realistic version of the second analysis as it works on a log file of events that happened in a system. The user of the tool must specify log points and the analysis calculates an over-approximation of actions taking place between log points. We evaluate our tool and find it to be a correct implementation of the framework developed.

1.2 Thesis Organization

In the next chapter we discuss some theoretical background that is necessary for following the rest of the thesis. We give an overview of the work done so far in statically analyzing the insider problem, define the notion of insider and insider threat, and discuss the Klaim family of process calculi.

Chapter 3 covers the development of the Insider Framework which is really the design of a tool for doing static insider analysis. In this chapter we will present a language for specifying systems and two insider analysis.

In Chapter 4 the Insider Framework will be extended to allow access points to be logged and we will give the final insider analysis.

Chapter 5 will cover the implementation of our tool and in Chapter 6 we will run the analyses on a few systems to demonstrate the correctness of our implementation. Finally Chapter 7 presents the conclusion of our work.

CHAPTER 2

Background

This chapter gives a brief theoretical background of the technologies we use in analyzing the insider problem. The goal of the chapter is to provide the reader with the necessary background to be able to follow the rest of the paper. The rest of this section is organized as follows: Section 2.1 gives an introduction to the Klaim family of process calculi, which is relevant for the analyses we define. Section 2.2 gives a definition of the terms ”insider” and ”insider threat”. Section 2.3 gives an overview of the work done so far in analyzing the insider problem, and finally Section 2.4 gives an overview of the Flow Logic framework [13].

2.1 Klaim

Distributed systems typically consist of a large number of heterogenous computational entities that execute components of applications. Components of distributed systems have to deal with unpredictable changes in the network environment over time, e.g., mobile components can disconnect from the network and reconnect later at a different node.

Klaim(**K***ernel* **L***anguage for* **A***gents* **I***nteraction and* **M***obility)* is a language designed to model distributed systems consisting of several mobile components that interact through multiple distributed *tuple spaces* [4]. The focus is on processes running in a WAN, where the overall structure of the WAN can change. Processes in Klaim can change the spatial structure of the network, and localities are first-class citizens that can be dynamically created and communicated over the network. A tuple space is a multi-set (the same element can occur several times in the set) of tuples that are sequences of information items. Tuples are anonymous and picked up from tuple spaces by means of *pattern-matching.* Interprocess communication is asynchronous; sender and receiver need not synchronize their actions.

The Klaim programming paradigm emphasizes a clear separation between the computational level and the network administrator level. Programmers design individual processes, but administrators design nets (more on nets and processes in the next section). Nets are clearly distinguishable from user processes, and modeled explicitly.

Klaim is a family of process calculi and the most basic version is cKlaim or *Core* Klaim, which can be seen as a variant of the π-calculus. μKlaim (*Micro* Klaim), is an extension of cKlaim with tuples and pattern matching. acKlaim, which will be used in Section 2.2, is an extension of μKlaim with access control primitives and a reference monitor semantics. There are many other extensions to cKlaim that we shall not cover here, but refer the interested reader to [4].

2.1.1 KlaimSyntax

The most basic version of Klaim is cKlaim; its syntax is listed in Figure 2.1. The syntax is composed of four syntactic categories, *Nets*, *Processes*, *Actions*, and *Templates.* Nets are finite collections of *nodes* where processes or data can be located. Nodes can be referenced by their locality, l, which represents the address of the node. Processes are the computational units in Klaim. The special process **nil** denotes the empty process that does nothing. Processes can run concurrently at the same location and they can execute four different actions. A Process can also invoke a process by its *process definition.* It is not possible to explicitly state the process definition, it is only assumed that every process identifier, A, has a defining equation of the form $\mathrm{A} \overset{\triangle}{=} \mathrm{P}$. There are four actions: **out**, which outputs a datum at the specified location, **in**, which uses a template to select data from a tuple space, **eval**, which is used to model mobility, and finally **newloc**, which creates a new location.

N	$::=$		NETS
		$l :: P$	single node
	$\mid$	$l :: \langle l' \rangle$	located datum
	$\mid$	$N_1 \mid\mid N_2$	net composition
P	$::=$		PROCESSES
		$\mathbf{nil}$	null process
	$\mid$	$a.P$	action prefixing
	$\mid$	$P_1 \mid P_2$	parallel composition
	$\mid$	A	process invocation
a	$::=$		ACTIONS
		$\mathbf{out}(l')@\ell$	output
	$\mid$	$\mathbf{in}(T)@\ell$	input
	$\mid$	$\mathbf{eval}(P)@\ell$	migration
	$\mid$	$\mathbf{newloc}(u)$	creation
T	$::=$		TEMPLATES
		ℓ	name
	$\mid$	$!u$	formal

Figure 2.1: cKlaimsyntax

2.2 Insider problem

Bishop [5] calls the "insider problem" the most difficult and critical problem in computer security to deal with. The term is also coined "the insider threat". An insider is especially dangerous because he has information and capabilities not known to external attackers. The insider can cause catastrophic damage as he has knowledge of both possible weak spots and assets in the company infrastructure.

It is important that we define what we mean by "insider threat" and "insider" to be able to define an analysis that can detect these kinds of threats. Bishop [5] introduces different definitions of the problem. The RAND report [2] defines the problem as *malevolent actions by an already trusted person with access to sensitive information and information systems*, and the insider as *someone with access, privilege, or knowledge of information systems and services.* Bishop defines the term insider and insider threat as:

Definition 2.1 (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 is 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.

From this definition it seems that the rules, R, consist of a set of access policies and a set of "trust" rules. The access granted to an actor could be limited to a special kind of circumstances, e.g., a janitor is not allowed to enter the server room in an organization unless there is fire in the server room. However, the restriction does not really limit him to go there anyway, even if there is no fire. He is "trusted" not to go there under normal circumstances.

2.3 Insider Work

This section gives the reader an overview of the work done so far in formally analyzing the insider problem. The main source of information is [16], which addresses analyzing the insider problem, and this section will explain the approach taken by the authors. The original work is based on the acKlaim process calculus, which is a member of the Klaim family of process calculi.

The goal of [16] is to analyze insider attacks by using program analysis techniques [3, 10, 11]. Traditionally, these kinds of attacks are analyzed by auditing log files *after* the attack has happened. In [16] a formal model of systems is developed that can describe real-world scenarios. By using this formal model we can specify the spatial structure, the actors, the access policies, and the data in the system we want to model. The formal model consists of two parts: an abstract, high-level system model based on graphs, and a process calculus called acKlaim that provides the formal semantics for the abstract model. acKlaim is an extension of μKlaim with access control primitives, named processes, and a reference monitor semantics.

In order to analyze insider attacks in a system the abstract system specification is mapped to an acKlaim program. While the authors of [16] describe the syntax and semantics of the acKlaim calculus, there is no description how systems (described using the formal model) are mapped to acKlaim programs. The reason for the mapping is that acKlaim provides a formal semantics that is used in the analysis, but the formal model of systems does not provide any semantics.

There are developed two analyses; The first is intended as a "before-the-fact" analysis that identifies week spots in systems. The analysis identifies which actions may be performed by whom, at which location, accessing which data. The goal is to discover a superset of incidents - before they occur. The second analysis is a control flow analysis that identifies insider threats given a sequence of actions for each actor in the system. The latter analysis is thus intended as an "after-the-fact" analysis where the given sequences of actions can be thought as actions reconstructed from a log file. The results of the latter analysis is more precise as the actor is limited by the actions in the sequence.

2.3.1 The Abstract System Model

The abstract system model is a collection of mathematical structures that is used to create an abstraction of a real-world system. The system model is high-level and is used to model the spatial structure of the system, as well as the location of data and actors in the system. When the abstract system has been mapped to acKlaim, the semantics of acKlaim uses the abstract system to evaluate the resulting net.

The first elements of the abstract model are the sets that define the spatial structure of a system.

Definition 2.2 (Infrastructure, Locations, Connections) $(\mathsf{Loc}, \mathsf{Con})$ represents an infrastructure which is a directed graph, where Loc is a set of nodes representing locations, and $\mathsf{Con} \subseteq \mathsf{Loc} \times \mathsf{Loc}$ is a set of directed edges representing connections between locations. $n_d \in \mathsf{Loc}$ is reachable from $n_s \in \mathsf{Loc}$, if there is a path $\pi = n_0, n_1, n_2, \ldots, n_k$ with $k \geq 1$ and $n_s = n_0, n_d = n_k$ and $\forall_i\ 0 \leq i \leq k-1 : n_i \in \mathsf{Loc} \wedge (n_i, n_{i+1}) \in \mathsf{Con}$.

Actors are the active entities in the system and can move along edges between nodes. Each actor is bound to a domain in which he can perform actions.

Definition 2.3 (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 2.4 (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.

Access control is modeled with a set of *capabilities* and *restrictions* that restrain the mobility of the actors and protect sensitive data. Capabilities are associated with actors, and restrictions are associated with locations and data.

Definition 2.5 (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 $\Phi_r : \mathsf{Cap} \rightarrow \{true, false\}$ checks whether the capability matches the restriction or not. $\mathcal{C} : \mathsf{Actors} \rightarrow \mathcal{P}(\mathsf{Cap})$ maps each actor to a set of capabilities, and $\mathcal{R} : (\mathsf{Loc} \cup \mathsf{Data}) \rightarrow \mathcal{P}(\mathsf{Res})$ maps each location and data item to a set of restrictions.

Finally, we give a definition of a system, which include all the elements in Definition 2.2 through Definition 2.5.

Definition 2.6 (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, $\mathcal{C} : \mathsf{Actors} \rightarrow \mathcal{P}(\mathsf{Cap})$ a mapping from actors to capabilities, $\mathcal{R} : (\mathsf{Loc} \cup \mathsf{Data}) \rightarrow \mathcal{P}(\mathsf{Res})$ a mapping from actors and locations to restrictions and for each restriction r, let $\Phi_r : \mathsf{Cap} \rightarrow \{true, false\}$ be a checker. Then we call $\mathcal{S} = \langle \mathcal{I}, \mathsf{Actors}, \mathsf{Data}, \mathcal{C}, \mathcal{R}, \Phi \rangle$ a system.

2.3.2 acKlaim Security Policies

As mentioned earlier, acKlaim is an extension of μKlaim with access control primitives, named processes, and a reference monitor semantics. This subsection introduces the security policies of acKlaim, as defined in [16]. Locations and data have to grant access to actors in order for the actor to perform actions in the system. The access can be granted in three different ways, and the reference monitor semantics ensures that these access policies are enforced. The access can be granted based on:

- **where** an actor is (the location the access request is coming from),
- **who** the actor is (based on the actor that is performing the access request),
- **what** the actor knows (based on an actor's knowledge of a secret, e.g., a secret key).

In acKlaim there are five different *access modes* (i, r, o, e, n), corresponding to the actions *destructive read a tuple* or *pickup a tuple*, *non-destructive read a tuple*, *output a tuple* or *produce a tuple*, *remote evaluate a tuple*, and *create a new location.*

$$\begin{array}{rcl} \pi \subseteq \mathsf{AccMode} & = & \{\mathsf{i}, \mathsf{r}, \mathsf{o}, \mathsf{e}, \mathsf{n}\} \\ \kappa \subseteq \mathsf{Keys} & = & \{\text{unique key identifiers}\} \\ \delta \in \mathsf{LocPolicy} & = & (\mathsf{Loc} \cup \mathsf{Actors} \cup \mathsf{Keys} \cup \{*\}) \rightarrow \mathcal{P}(\mathsf{AccMode}) \end{array}$$

Figure 2.2: Access control in acKlaim

The special symbol * is used to define default access policies (that are allowed for locations, keys, and actors).

2.3.3 acKlaim syntax

The syntax of acKlaim is similar to the μKlaim syntax, extended with access policies, a set of keys for each processes and named processes. The syntax consists of three major classes of syntactic constructs: *nets, processes and actions*, as shown in Figure 2.3 and Figure 2.4.

Nets are finite collections of *nodes* which can contain processes and data. A node has a locality, l, and either a *process* or a *datum.* If the node is a process node, the node is annotated with a LocPolicy, the name of an actor, and the actors keys. On the other hand, if the node is a datum node, it is annotated with a policy from LocPolicy, and contains a datum element. Finally, nets can be the composition of two nets. Note that nets do not give any information on *how* nodes are interconnected.

Processes are the active computational units of acKlaim. Processes execute by performing sequences of actions, and processes run concurrently at either the same location or at different locations in the system. A process can be the **nil** process that does nothing, or a *sequence of actions* built up from the **nil** process. A process can be the *composition* of two or more processes and can contain an *invocation* to a named process definition. Using the syntax given in Figure 2.3 there is no way to specify process equations explicitly, and thus define a process definition these equations are assumed to be available to the semantics.

There are five actions available. The **out** action produces a tuple and places it at a given location. The **in** action destructively reads a tuple from the specified location. The **read** action reads a tuple from a location in a non-destructive manner. The **eval** action is used for processes to spawn a new process at a given location, it is used to model mobility in the system. Finally the **newloc** action creates a new location.

Tuples are the communicable objects in acKlaim and are sequences of actual fields. The actual fields are expressions, localities, and locality variables. The expressions are deliberately not specified but contain at least values, V, and value variables, x. Templates are sequences of actual and formal fields. Formal fields are variables with an exclamation mark like ($!x$, $!u$) are used to bind values. There are two ways to bind a variable in acKlaim; the action $\mathbf{in}(!x)@\ell.P$ binds the variable x in P, and $\mathbf{newloc}(u)@\ell.P$ binds the variable u in P. When tuple spaces are read by using either **in** or **read**, a tuple that matches the input pattern (template) is read in an arbitrary way. For actions **in** and **out** the templates must be evaluated before they are added to a tuple space. The template evaluation consists of computing the value of the expressions occurring in the template. Templates with variables in actual fields cannot be evaluated. We use $[\![T]\!]$ to denote an evaluated template.

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

Figure 2.3: 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	$::=$	$e \mid \ell$	tuple fields				

Figure 2.4: Syntax for tuples and templates.

2.3.4 acKlaim semantics

The operational semantics for acKlaim is given as a structural congruence on nets and processes and with a *reduction relation* over nets. The structural congruence $\equiv$ identifies, which nets intuitively represent the same net. The structural congruence relation simplifies presentation of the semantics and the reasoning about processes. The relation is defined in Figure 2.5.

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

Figure 2.5: Structural congruence on nets and processes.

The (Com) rule represents the commutative law for nets - as a net does not give any explicit structure of nodes, it does not matter in which order they are presented in the net. The (Assoc) rule represents the associativity law for nets - it says that the order of evaluation does not matter. That makes perfect sense, as all processes in the net run concurrently. The (Abs) rule says that it is always safe to add/remove a concurrent running **nil** process to a process P. The (Inv) rule represents process invocation and says that when invoking a process named A, the body of the process A, is substituted out for the name. The (Clone) rule says that we can split a process that is running concurrently as one process into two processes running at the same location with the same security labels, and the other way around.

Access control is enforced by a *reference monitor* that is embedded in the operational semantics of the calculus. The reference monitor checks that access to data and localities is in accordance with their access policy. The reference monitor is defined in Figure 2.6.

The reference monitor has three parts: The function *grant* (R1), the judgement $\succ$ (R2), and the judgement $\rightsquigarrow$ (R3). The *grant* function checks whether an actor n at location l knowing κ may perform the action a on location l'. The action a is allowed at l', if any one of these conditions hold:

- the actor n is explicitly allowed to perform a at l',

(R1)

$$\text{grant} \;:\; \mathsf{Names} \times \mathsf{Loc} \times \mathsf{Data} \times \mathsf{AccMode} \times \mathsf{Loc} \longrightarrow \{\texttt{true}, \texttt{false}\}$$

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

(R2a)

$$\frac{l = t}{\langle \mathcal{I}, n, \kappa \rangle \succ (l, t)}$$

(R2b)

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

(R3)

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

Figure 2.6: Reference monitor for access control

- an actor knowing κ is allowed to perform a at l',
- an actor located at location l is allowed to perform a at l'.

The judgement $\succ$ (R2) decides whether an actor n at location l can reach a location t by following the edges in the infrastructure $\mathcal{I}$. An actor can reach a location t from l if he is allowed to perform the action *eval* along some path starting from l and finally reaching t. The actual path is not important, only that there *exists* a path.

The judgement $\rightsquigarrow$ (R3) combines the other two rules and says that an actor n in infrastructure $\mathcal{I}$ and knowing κ can perform action a at t if he is granted access by the *grant* function, and he is able to reach the location t from his current location l by a sequence of *eval* actions.

The reduction relation $\succ\!\!\longrightarrow_{\mathcal{I}}$ is defined in Figure 2.7 and is specified as a small step operational semantics. The boxed part of each rule is the access-control check performed by the reference monitor. The spatial structure of the system is represented as $\mathcal{I}$, and is assumed to be given.

The rule (OUT) says that an actor n can output a datum t at location l', if there exists a location l' with a process P', and the actor n is allowed to perform the action at l'. The rule (IN) says that an actor n can pick up a datum et at location l' if it matches the template T. The datum node is then reduced to a **nil** process node l', so that the location is not lost if there are no processes at

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

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

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

(EVAL)
$$\frac{\boxed{\langle \mathcal{I}, n, \kappa\rangle \rightsquigarrow \langle l, l', \mathsf{e}\rangle}}{\begin{array}{c} l ::^{\delta} [\mathbf{eval}(Q)@l'.P]^{\langle n,\kappa\rangle} \mid\mid l' ::^{\delta'} [P']^{\langle n',\kappa'\rangle} \rightarrowtail_{\mathcal{I}} \\ l ::^{\delta} [P]^{\langle n,\kappa\rangle} \mid\mid l' ::^{\delta'} [Q]^{\langle n,\kappa\rangle} \mid\mid l' ::^{\delta'} [P']^{\langle n',\kappa'\rangle} \end{array}}$$

(NEW)
$$\frac{l' \not\in L \qquad \lfloor l' \rfloor = \lfloor u \rfloor}{\begin{array}{c} L \vdash l ::^{\delta} [\mathbf{newloc}(u^{\pi} : \delta').P]^{\langle n,\kappa\rangle} \rightarrowtail_{\mathcal{I}} \\ L \cup \{l'\} \vdash l ::^{\delta[l' \mapsto \pi]} [P[l'/u]]^{\langle n,\kappa\rangle} \mid\mid l' ::^{\delta'[l'/u]} [\mathbf{nil}]^{\langle n,\kappa\rangle} \end{array}}$$

(PAR)
$$\frac{L \vdash N_1 \rightarrowtail_{\mathcal{I}} L' \vdash N_1'}{L \vdash N_1 \mid\mid N_2 \rightarrowtail_{\mathcal{I}} L' \vdash N_1' \mid\mid N_2}$$

(STRUCT)
$$\frac{N \equiv N_1 \quad L \vdash N_1 \rightarrowtail_{\mathcal{I}} L' \vdash N_2 \quad N_2 \equiv N'}{L \vdash N \rightarrowtail_{\mathcal{I}} L' \vdash N'}$$

Figure 2.7: Operational Semantics for acKlaim

the location.

The rule (READ) is similar to the (IN) rule. It adds the datum et to the tuple space of the actor n if the datum et matches the template T. The action does not remove the datum from the location. The **eval** action takes a process Q as an argument and is used to spawn a new process at given location. The (EVAL) rule does this by adding the new process to the system, at the given location l', with the same access policies as were defined for l'. The **newloc** action creates a new location u, and it takes three parameters: The variable u which will hold a reference to the new location, π which is the change in the current locations access policy that applies to the new location, and finally δ' which is the access policy at the new location. The new location gets an arbitrary name l' that is substituted for the variable u in the process. The (PAR) rule states that if a part of a net is reduced, the whole net is reduced accordingly. This makes the rules for actions able to "focus" on small parts of the net. The last rule (STRUCT) relates the structural congruence and the reduction relation by saying that all structural congruent nets can make the same reduction steps.

The final part of the semantics is the rules for *template maching.* The *match* function takes two arguments and returns a substitution σ. The substitution formalizes how a value should be substituted in a process term. For example, $P[l'/u]$ expresses that all occurrences of u should be substituted out for l' in the process term P. The *match* rule is defined in Figure 2.8.

$$match(V, V) = \epsilon \qquad match(!x, V) = [V/x] \qquad match(l, l) = \epsilon$$

$$match(!u, l') = [l'/u] \qquad \frac{match(F, ef) = \sigma_1 \qquad match(T, et) = \sigma_2}{match((F, T), (ef, et)) = \sigma_1 \circ \sigma_2}$$

Figure 2.8: Semantics for template matching

2.3.5 Analysis

As mentioned before, an abstract model of a system is mapped to an acKlaim program in order to analyze the insider properties of the system. In this section two analysis on systems will be described. The first analysis determines which locations in a system an actor with name n and keys κ can reach from location l. This analysis gives us a map over which locations and data an insider can reach. This analysis would be useful as a "before-the-fact" analysis, to identify vulnerabilities in the system. The analysis is graph-based and is defined on the abstract spatial structure of the system, it does not use the acKlaim semantics at all.

The second analysis is a control-flow analysis of the actors in the system. It determines which data a specific actor may read and which location he may reach, given a process definition for the actor. This analysis could e.g., be based on log-files after an attack had been made. This analysis is done on an acKlaim program where process equations are "plugged into" the acKlaim program to model the behavior of actors.

2.3.5.1 Reachability Analysis

The first analysis is a reachability analysis where we are interested in a subset of locations a given actor can reach. The analysis first calculates the set of locations that each individual actor in the system can reach, by placing the actor at each location in the system and then calculating the set of locations the actor can reach from there. Then the result is combined to find the set of locations the actor can possibly reach in the system. The result of the reachability analysis 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. The analysis can be split into three parts:

1. For a given actor, find the set of locations he can reach for a given location,
2. for a given actor do 1) for each location in the system and create a union of the results,
3. for all locations in 2) create a mapping from actions to locations that represent the actions the actor can perform at those locations.

Each of the three parts have their own algorithm as described in Algorithm 1 - Algorithm 3.

Algorithm 1 checkloc $(\mathsf{Names} \times \mathsf{Loc} \times \mathsf{Keys} \times (\mathsf{Con} \times \mathsf{Loc}) \rightarrow \mathcal{P}(\mathsf{Loc}))$

```
checkloc$(n, l, \kappa, \mathcal{I})$ =
for all $(l, l') \in \mathsf{Con}$ do
    if $\langle \mathcal{I}, n, \kappa \rangle \succ (l, l') \vee \mathrm{grant}(n, l, \kappa, \mathsf{e}, l')$ then
        return $\{l'\} \cup \mathrm{checkloc}(n, l', \kappa, \mathcal{I})$
    end if
end for
```

Algorithm 2 checksys $(\mathsf{Names} \times \mathsf{Keys} \times (\mathsf{Con} \times \mathsf{Loc}) \rightarrow \mathcal{P}(\mathsf{Loc}))$

```
checksys$(n, \kappa, \mathcal{I}) = \bigcup_{l \in \mathsf{Loc}} \mathrm{checklock}(n, l, \kappa, \mathcal{I})$
```

Algorithm 3 checkdata $(\mathsf{Names} \times \mathsf{Keys} \times (\mathsf{Con} \times \mathsf{Loc}) \rightarrow \mathcal{P}(\mathsf{AccMode} \times \mathsf{Loc}))$

```
checkdata$(n, \kappa, \mathcal{I})$ =
$\bigcup_{l \in \mathrm{checklock}(n, l, \kappa, \mathcal{I})} \{(a, l) | \exists\ a \in \mathsf{AccMode}, (l, l') \in \mathsf{Con} : \mathrm{grant}(n, l, \kappa, a, l')\}$
```

2.3.5.2 Control-Flow analysis

The second analysis takes into account what the actor actually does in the system. The analysis works on the acKlaim program and a process definition is substituted for the process call in the program. This analysis computes a conservative approximation of all the possible flows into and out of all the tuple spaces in the system and what values each variable can possible have during execution. The analysis is specified in the Flow Logic framework, [13]. It is specified with a number of judgements, one for each syntactic category. The judgements determine whether or not a given analysis estimate is valid. The Flow Logic specification does not directly calculate the analysis estimate but can be used to generate constraints that can be solved by a constraint solver to produce the analysis estimate. In the original work [16] only the Flow Logic specification is given which is a common practise in language-based research. The Flow Logic specification is modeled in Figure 2.9 and Figure 2.10.

$$
\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 \mid\mid 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 \quad \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') : (\langle \mathcal{I},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathsf{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') : (\langle \mathcal{I},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathsf{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') : (\langle \mathcal{I},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathsf{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},\mathsf{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}
$$

Figure 2.9: Flow Logic Specification for Insider Analysis.

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

Figure 2.10: Flow Logic Specification for Insider Analysis.

2.4 Flow Logic

This section gives an overview of the Flow Logic framework, which is an approach to static analysis. Flow Logic is a formalism for static analysis based on logic systems. It is similar to type systems and structural operational semantics. Flow Logic focusses on specifying what it means for an analysis estimate to be acceptable for a program. A Flow Logic specification is a set of rules that specify what an acceptable analysis estimate should look like. For a given language there is one judgement for each syntactic category and one rule for each syntactic construct. There is a clear distinction between specifying what is an acceptable analysis estimate and how to compute one, so a Flow Logic specification does not say anything about how to compute an analysis estimate. A schematic view of a Flow Logic specification and its use can be viewed in Figure 2.11.

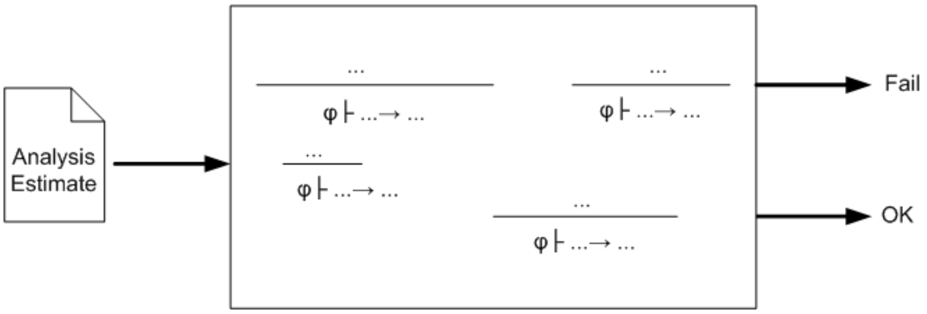


Figure 2.11: Flow Logic Specification

It is possible to specify Flow Logic specifications at different levels of abstractions, just as semantic specifications can be "small step" and "big step". There are two sets of styles for presenting Flow Logic specifications: *abstract* versus *compositional*, and *succinct* versus *verbose*. A compositional specification is syntax-directed and is closer to an implementation than the succinct, like the succinct is closer to the semantics of the language.

Flow Logic specifications can be used to generate analysis estimates. The implementations generate constraints and then a constraint solving algorithm solves these constraints and calculates a analysis estimate. Figure 2.12 gives an overview of this procedure.

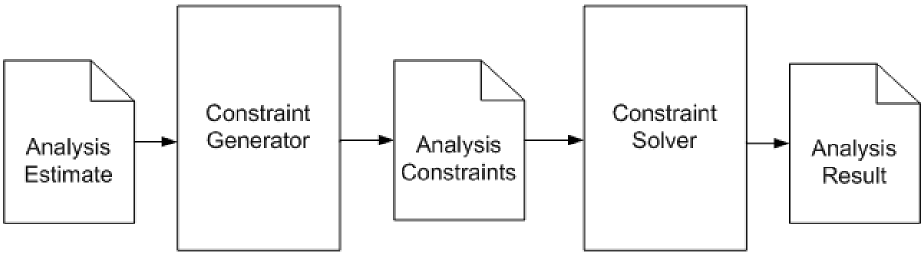


Figure 2.12: Using a Flow Logic Specification to Generate an Estimate

We provide Flow Logic specifications for our analysis but we do not use them to generate constraints and solve them. We have decided to produce our analysis results using graph-based algorithm and consider the implementation of constraint based solver to be future work. We refer to [13, 8] for more details on the Flow Logic framework.

Chapter 3

The Insider Framework

This chapter describes our own work on analyzing the insider problem. We will follow closely the method as described in [16] and this section contains pieces from the original work as well as extensions to the original work. The goal of this section is to analyze and design an implementable *framework* for doing insider analysis. The first version of the framework we will extend [16] such that data can be used as keys, and data has security annotations. We will also add actions to *encrypt* and *decrypt* data.

3.1 Framework Requirements

Our goal is to be able to analyze insider-threats in real-world systems. The system model should contain interconnected localities, actors that can move around in the localities and perform actions, and data that can be moved around and exchanged between actors. We would like to have some kind of access-control mechanism to limit the mobility of actors and to control the access to data. The system model should be high-level and easy to describe and understand for people with limited process calculus knowledge.

We are interested in two kinds of insider analyses, an analysis of the *spatial* structure of a system and an analysis that reconstructs sequences of actions based on log files recorded in the system. It should thus be possible to log each access-control check in the system. When a system is modeled, access control points should be marked if they should leave an entry in the log or not. We will not focus on the logging of access-control points in the first version of our *framework*, but add it to the framework in Chapter 4. First we want to focus on actors exchanging information, e.g. if two actors are in the same room they could possibly exchange some information, even knowledge of keys that can be used to encrypt and decrypt data or access locations.

Finally we would like to implement a tool that can take a system specification as an input and perform the desired analysis on it and report any insider threats that can be found. The tool should also be able to take a log file as input and use the log file to reconstruct a sequence of actions that possibly took place in the system.

3.2 Insider Framework

We have chosen an approach that is similar to the approach taken in [16], i.e., to map a specification of an abstract, high-level system to a process calculus program and then run the analysis on the program. The process calculus we use is insCalc which is inspired by acKlaim. The reason for the mapping is that insCalc has a formal semantics that is used to specify the analysis. Our approach is more detailed than in [16] as we want do describe an implementable way of performing insider analysis. The sequence of actions (pipeline) for performing insider analysis is shown in Figure 3.1.

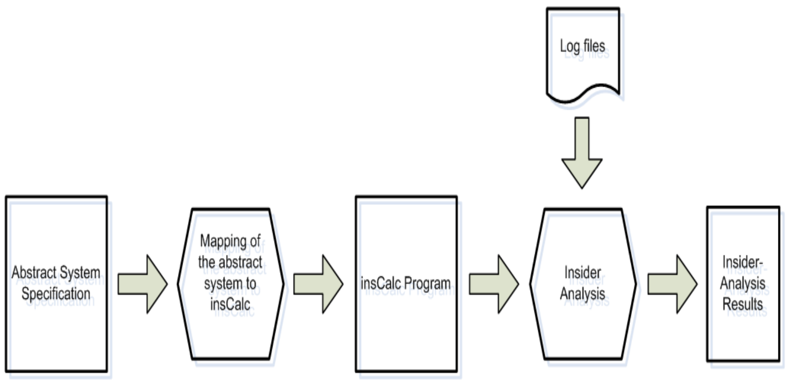

Log files
Abstract System Specification
Mapping of the abstract system to insCalc
insCalc Program
Insider Analysis
Insider-Analysis Results

Figure 3.1: Sequences of actions during insider analysis

The term *insider framework* is used for the collection of technologies used in analyzing the insider problem and it consists of the following elements:

- **An abstract system model**: An abstraction used to model the systems we want to analyze as mathematical objects,
- **Syntax for specifying the abstract systems**,
- **Syntax of the insCalc process calculus**,
- **A mapping from abstract systems to insCalc programs**: A formal mapping of abstract system models to insCalc,
- **Semantics of the insCalc process calculus**,
- **Analysis of systems**: Analysis that finds insider threats in the systems,
- **Implementation of a system**: A tool for analyzing models for insider threats.

3.3 Running Example

To make the discussion more concrete, new concepts will be demonstrated by a running example system. The running example is inspired by [16] and is shown in Figure 3.2. The system consists of four physical locations, a janitor's workshop, an user office with a computer and a waste basket, a printer room with a printer, computer and waste basket, and finally a hallway interconnecting the locations. The computers and the printer are connected, so the computers can send data to the printer and between each other. There are two actors in the system, a user and a janitor. Initially they are located in the office and in the janitor's workshop respectively. The actors in the system should have to log into the computer, this constraint is modeled in Figure 3.2 by a lock on each computer in the system. For security reasons there are cipher-locks on the entrance of the office and printer room, and each user has to know some PIN-code to be able to access these rooms. The janitor's workshop is locked with a "normal" lock, modeled in Figure 3.2 with a lock on the door. The janitor should have a key to open his own workshop, and the user should have a PIN-code to access the office and the printer room. Data in this example could be data coming from the printer, or lying in the waste basket. The janitor also has the PIN-code to enter the printer room, but we assume that he is "trusted" to not go to the printer room unless there is something broken; e.g., a new toner needed for the printer or a light bulb broken.

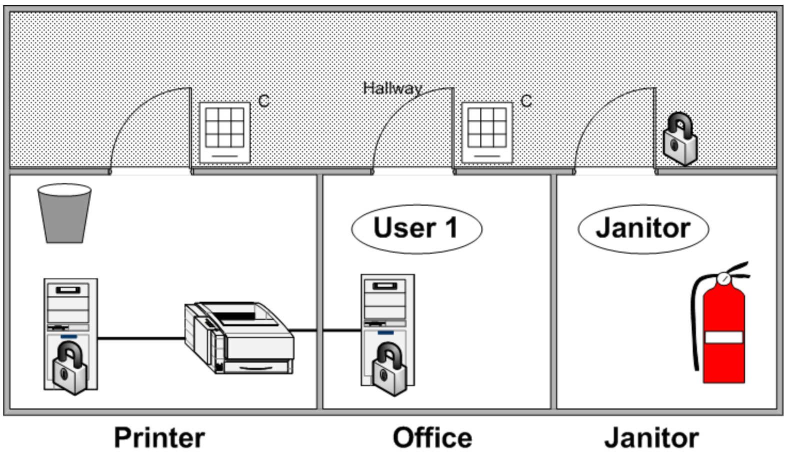


Figure 3.2: The running example

3.4 Abstract systems

The first element in the insider framework is the abstract system model. We define the model as a collection of mathematical constructs for creating *an abstraction* of real-world systems. The reason for defining these mathematical constructs is that we are going to need them later when we specify the semantics of insCalc, our variation of the acKlaim calculus. The mathematical objects are also convenient for an internal representation of our system when implementing a tool for insider analysis.

The abstract system models real-world systems, e.g., with physical localities, interconnected computers, actors that can move around in the physical localities, and data that can be carried by actors or left at both computers or physical localities. It should be possible for actors to exchange data. On top of this there is a fine-grained access control mechanism that limits the mobility of actors, and protects sensitive data. In the abstract system there is no means for explicit movement of actors, but only a means of specifying what the initial structure of the system looks like. That is, an abstract system specifies where actors are located, where data is located, how locations are interconnected, and access control in the system but there is no way to specify that an actor should move to another location or do anything else.

There is not much difference in our definitions from [16], but we will repeat the definitions here for completeness. The first part of the system is an *infrastructure*, which is a directed graph that models the spatial structure of the system. The graph is modeled as a set of nodes that represents *locations* and a set of directed edges that represents *connections*.

Definition 3.1 (Infrastructure, Locations, Connections) $(\mathsf{Loc}, \mathsf{Con})$ represents an infrastructure, which is a directed graph, where Loc is a set of nodes representing locations, and $\mathsf{Con} \subseteq \mathsf{Loc} \times \mathsf{Loc}$ is a set of directed edges between nodes representing connections between locations. $n_d \in \mathsf{Loc}$ is reachable from $n_s \in \mathsf{Loc}$, if there is a path $\pi = n_0, n_1, n_2, \ldots, n_k$ with $k \geq 1$ and $n_s = n_0, n_k = n_d$ and $\forall_i\ 0 \leq i \leq k-1 : n_i \in \mathsf{Loc} \wedge (n_i, n_{i+1}) \in \mathsf{Con}$.

Actors are the active entities in the system and can move along edges between nodes. Actors model persons moving around in physical locations, or computer programs that migrate from one network location to the next. We therefore say that actors move in a particular *domain*, and we typically make a distinction between the "physical domain" and the "digital domain". PC's are at the boundary between these two domains, as an actor could start a program on a computer that then migrates to another computer using the underlying network. Actors cannot, of course, move across domains boundaries, but data can do so as it might be uploaded to a computer, read from a screen or printed out. *Data* can be located at locations and actors. It can be produced, picked up, and read by actors.

Definition 3.2 (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. Finally $\mathcal{G} : \mathsf{Actors} \rightarrow \mathsf{Loc}$ defines the location at which each actor is located.

Definition 3.3 (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.

Access control is modeled with a set of *capabilities* and *restrictions* that restrain the mobility of the actors and protects sensitive data. Capabilities are associated with actors, and restrictions are associated with locations and data. Capabilities of an actor is something that enables him to get access to data or locations. Restrictions of data and locations are policies that limit access to the resource.

Definition 3.4 (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 $\Phi_r : \mathsf{Cap} \to \{true, false\}$ 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.

Finally, we define a system which includes all the elements in Definition 3.1 through Definition 3.4.

Definition 3.5 (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, $\mathcal{D} : \mathsf{Loc} \to \mathsf{Dom}$ a mapping from locations to domains, Cap a set of capabilities, Res a set of restrictions, $\mathcal{C} : \mathsf{Actors} \to \mathcal{P}(\mathsf{Cap})$ a mapping from actors to capabilities, $\mathcal{R} : (\mathsf{Loc} \cup \mathsf{Data}) \to \mathcal{P}(\mathsf{Res})$ a mapping from actors and locations to restrictions, $\mathcal{G} : \mathsf{Actors} \to \mathsf{Loc}$ a mapping from actors to locations and for each restriction r, let $\Phi_r : \mathsf{Cap} \to \{true, false\}$ be a checker. Then we call $\mathcal{S} = \langle \mathcal{I}, \mathsf{Actors}, \mathsf{Data}, \mathcal{D}, \mathcal{G}, \mathcal{C}, \mathcal{R}, \Phi \rangle$ a system.

3.5 Access Policies

In the abstract system, access control is modeled by a set of capabilities and a set of restrictions. We now specify what the access policies and restrictions should look like. The definitions presented in this chapter are a part of the insCalc syntax presented in Section 3.7.

We want to limit the access of actors to locations and data. This means that a location or datum has to explicitly grant access to an actor in order for the actor to perform actions on the datum or in the location. There are six different *access modes* (i, r, o, e, m, d) corresponding to the actions *destructive read a tuple* or *pickup a tuple*, *non-destructive read a tuple*, *output a tuple* or *produce a tuple*, *remote evaluate a tuple*, *move*, and *decrypt data.* These modes are similar to the modes in the original work on the insider problem, but we have removed the *new location*, as we do not want to model actors creating new locations in the system, and added the move and the decrypt access mode. The reason we removed the *new location* is that it does not seem realistic that actors go around and create new locations, locations are a static part of real-world systems. The first five access modes apply to locations only, and the last one is intended for data only. We do not have an access mode for *encryption* as we do not find it realistic that a datum could make such a restriction on itself.

The access to locations and data can be granted in three different ways, based on:

- **Where** an actor is (the location the access request is coming from),
- **Who** the actor is (based on the actor that is performing the access request),
- **What** the actor knows (based on an actors knowledge of a secret, e.g., a key).

$$\begin{array}{rcl}
\pi_\ell \subseteq \mathsf{LocAccMode} & = & \{\mathsf{i}, \mathsf{r}, \mathsf{o}, \mathsf{e}, \mathsf{m}\} \\
\pi_\delta \subseteq \mathsf{DataAccMode} & = & \{\mathsf{d}\} \\
\kappa \subseteq \mathsf{Data} & = & \{\text{data used as keys}\} \\
\delta \in \mathsf{LocPolicy} & = & (\mathsf{Loc} \cup \mathsf{Actors} \cup \mathsf{Data} \cup \{*\}) \rightarrow \mathcal{P}\{\mathsf{LocAccMode}\} \\
\rho \in \mathsf{DataPolicy} & = & (\mathsf{Loc} \cup \mathsf{Actors} \cup \mathsf{Data} \cup \{*\}) \rightarrow \mathcal{P}\{\mathsf{DataAccMode}\}
\end{array}$$

Figure 3.3: Access control in the abstract system

Figure 3.3 shows the set of access modes for both locations and data and how policies are defined. The special element * allows to specify a set of access modes that are allowed by default. The Data element in LocPolicy and DataPolicy is used to model keys, so there is no distinction between data and keys. As soon as an actor picks up, reads or outputs a datum, it will be available to him as a capability. This is different from [16], where there was a special domain for keys and keys where fixed for each actor. There are two sets of access policies one for locations, LocPolicy, and one for data, DataPolicy.

Locations define access policies to limit the movement of actors or protect access to data. The access modes in the set LocAccMode are intended to be used for specifying access policies for locations only. The location defines a policy that explicitly allows actors, locations, or data access to the location and its data. The r access mode is used to allow that data within the location can be read, this would be an obvious choice for data that is written on a blackboard or data that cannot move away from the location. If data *can* be moved, the i access mode is a better choice. It may seem strange at first that a location could grant a key access to read or pick up data, but imagine a vault that contains data and is protected with a PIN-code. The policy for the vault would be something like

$$\{PIN\_CODE : i, r\}$$

Anyone with the knowledge of the PIN-code is allowed to open the vault and retrieve the data. The o access mode is used to allow output of data at a location.

Most locations would probably allow a actor to output data, but a computer or a terminal that is only used to display data might not allow anyone to insert new information. The e access mode is used to allow actors to spawn a new process in the system, and the m access mode controls the movement of actors into the location. Locations that model, for example, PC's and other devices would not allow actors to move into them, but only to retrieve data from them.

Access policies for data are the modes in the DataAccMode set. Currently there is only one mode in the set. Once an actor has picked up or read data, he is free to move around with that data, so the only restriction the data can impose is encryption. It does not make sense to have the other access modes for data access. An access policy for data ρ can limit access to it by requiring the actor to be at a specific location. That may seem strange at first, but imagine that we want to model a decryption device that must be given encrypted data and can then output the original data. In this case it makes perfect sense to model this with a location restriction.

3.6 System Specification Language

The abstract system specification described in Section 3.4 is not well suited for implementation (as it is a collection of mathematical definitions), thus we need to define some kind of language for specifying systems. The language will make it possible for the one implementing the framework to parse a specification written in the language, and create an internal representation of systems, similar to the definitions in Section 3.4.

This section presents a language for specifying systems. Although an abstract specification will be mapped to an insCalc program, the syntax of the language should be as close to the abstract specification as possible but not like insCalc. The reason for this is that the user designing the system abstraction should not have to know anything about insCalc in order to use the analysis tool. The syntax of the specification language is given in Figure 3.4 and Figure 3.5.

Like the specification from Section 3.4 a system is composed of four major syntactic categories: *locations*, *connections*, *actors*, and *data*. *Locations* consists of a location name along with a list of restrictions that the location makes on actions performed on it. Each restriction is a name (location, actor, data, or star) and a list of actions that the name is allowed to perform at the location. Each location also specifies the 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 locations is not allowed to be empty as any interesting system must have

Spec	::=	`locations:`*Locations*;	
		`connections:`*Connections*;	
		`actors:`*Actors*;	
		`data:`*Data*;	specification
Locations	::=	*Location*	a single location
	|	*Location*, *Locations*	a list of locations
Location	::=	Loc{*LocPolicyList*}(Names)	a location
LocPolicyList	::=	ϵ	empty policy list
	|	*Policy*	a single policy
	|	*Policy*; *LocPolicyList*	a policy list
Policy :	::=	Names : *LocAccList*	
	|	* : *LocAccList*	
LocAccList	::=	ϵ	empty access list
	|	*LocAccMode*	a single access mode
	|	*LocAccMode*; *LocAccList*	an access mode list
LocAccMode	::=	`i`	destructive read
	|	`r`	non-destructive read
	|	`o`	output a datum
	|	`e`	spawn a process
	|	`m`	move

Figure 3.4: System Specification Language, part I

at least one location. The list of restrictions for a location, however, is allowed to be empty on a location and the empty meaning that no restrictions are imposed on the access of that location, written as: `MyLocation{}`. To model that *no* access is allowed by anyone, the list of access modes should be left empty as in `MyLocation{*:}`.

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. The connections are only in one direction not both directions. Both the locations in a connection must be defined for the connection to be well-formed.

Actors is a list of actor names and the name of the location they are located at initially. 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 previously defined in the list of locations.

$Connections$	$::=$	ϵ	empty connection list
	$\mid$	$Connection$	a single connection
	$\mid$	$Connection, Connections$	a list of connections
$Connection$	$::=$	Loc->Loc	a one way connection
$Actors$	$::=$	ϵ	empty actors list
	$\mid$	$Actor$	a single actor
	$\mid$	$Actor, Actors$	a list of actors
$Actor$	$::=$	Actors@Loc	an actor at a location
$Data$	$::=$	ϵ	empty data list
	$\mid$	$Datum$	single piece of datum
	$\mid$	$Datum, Data$	data list
$Datum$	$::=$	Data$\{DataPolicyList\}$@Loc	data at a location
	$\mid$	Data$\{DataPolicyList\}$@Actors	data at an actor
$DataPolicyList$	$::=$	ϵ	empty policy list
	$\mid$	$DataPolicy$	a single policy
	$\mid$	$DataPolicy; DataPolicyList$	a data policy list
$DataPolicy:$	$::=$	Names : $DataAccMode$	
	$\mid$	* : $DataAccMode$	
$DataAccMode$	$::=$	ϵ	empty access
	$\mid$	d	decrypt

Figure 3.5: System specification language, part II

Data is a list of data elements annotated with access restrictions and information on where they are located (either at an actor or at a location). The structure of the data is one-dimensional, a single string. This could be extended to a more complex tuple structure with nested tuples and so on, but we decided to keep the data format simple, for ease of presentation, as complex data does not give a more detailed analysis result. The same rules apply for the policies of data as well as the policies of locations. 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 * element, the datum is unreadable for every actor in the system. The only access restriction there is for data is *decryption*. Any actor is free to pick up or read data (as long as he has location access to it) but to get the information that the data holds he will have to have the necessary access to be able to decrypt it. An actor can of course only pick up or read data that is located at a location, not data that another actor holds.

```
locations: HALL{*:m}(phys), JAN{key1:m}(phys),
        OFF{1234:m}(phys),
        SRV{4321:m}(phys), WASTE{SRV:i,r,o}(phys),
        PC1{PC2:e; pass:e,i,r,o}(dig),
        PC2{PC1:e; pass:e,i,r,o}(dig),
        PRT{PC1:o; PC2:o; SRV:i,r}(dig);
connections: HALL->JAN, JAN->HALL,
        HALL->OFF, OFF->HALL, HALL->SRV,
        SRV->HALL, OFF->PC1, SRV->PC2,
        SRV->WASTE, SRV->PRT, PC1->PC2,
        PC2->PC1, PC2->PRT, PC1->PRT;
actors: USER@OFF, JANITOR@JAN;
data: 1234{}@USER, key1{}@JAN, 4321{}@JAN,
      4321{}@USER, pass{}@USER;
```

Figure 3.6: The running example modeled in the system specification language

Figure 3.6 shows the running example specified in the syntax just described. The specification is much more detailed than the initial system viewed in Figure 3.2, as it shows which restrictions each location, and data imposes. The normal key lock for the janitor's workshop is controlled by a key that the janitor currently holds and the cipher lock for the office is controlled by a PIN-code that the user has. There is also a PIN-code for the printer room and both the janitor and the user hold the PIN-code to this location. The `m` mode is used to model migration or movement of actors, so `HALL{*:m}` means that every actor is allowed to move to the hall. Both the PCs are protected by a password that only the user knows, so that only the user in the system can log on them, but not the janitor. The four data elements in the system, `1234`, `key1`, `4321`, and `pass` are all public and are used as keys. The example can now be visualized as shown in Figure 3.7.

3.7 insCalc Syntax

In the previous sections we have described what systems should look like, and using the systems specification language it should be easy to model those systems. We now move our attention to the insCalc process calculus, as we want to map the abstract systems to insCalc analyse on the generated programs. The syntax of insCalc is similar to acKlaim, but it has a more simple pattern matching mechanism and some different actions.

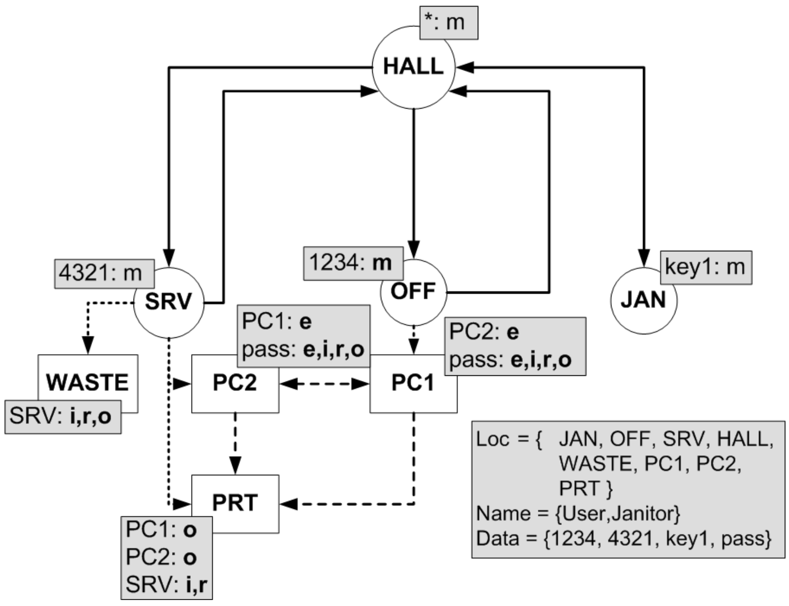


Figure 3.7: The running example as a graph

The syntactic categories are the same as for the Klaim family, i.e. *localities*, *nets*, *processes*, and *actions*.

Localities can be a name or a locality variable. The locality variable can be used to communicate a name of a location between actors, e.g., actors could decide to meet at a given location and one of the actors communicates that to the other actor by giving him a name of a location. Actors communicate by outputting data, picking data up, or reading it.

Nets are a collection of nodes that represent the localities in the system. A system is modeled by a net with nodes that hold either a process or a datum. If a node has a non-**nil** process, the process is tagged with a name and a tuple of data that can be used as keys. If the node contains a datum, the datum is tagged with a security policy that restricts access to it. A locality containing the **nil** process models a location with no actor and no data. It is not possible to model the spatial structure of a system with the syntax of nets, and in the semantics the spatial structure is assumed to be specified separately.

Processes are the active computational units of insCalc and they have the same structure as in acKlaim. Processes execute by performing sequences of actions. Processes run concurrently at the same location or at different locations in the system. A process can be the **nil** process that does nothing, or a *sequence of actions* built up from the **nil** process. A process can be the *composition* of two or more processes and can contain an *invocation* to a named process definition. There is no way to specify process equations, and thus define a process definition, with the syntax given in Figure 3.8, these equations are assumed to be available to the semantics.

There are seven *actions* available. The **out** action produces a tuple and places it at a given location. The **in** action destructively reads a tuple from the specified location. The **read** action reads a tuple from a location in a non-destructive manner. The location accessed by **out**, **in** and **read** is the current location of the actor, or one of its neighbor-locations. As a result an actor cannot magically output, read, or take a datum from a location that is not 'close' to him. The **eval** action is used to spawn new processes in the digital domain. Programs or human actors can spawn a new process that runs in the digital domain. The movement of actors is realized by the **move** action, that moves the actor to the specified location.

It should not be possible for actors, in the physical domain, to perform the **eval** action in the physical domain but only *from* the physical domain *to* the digital domain. Actors in the digital domain should likewise not be able to perform the **eval** action from the digital domain to the physical domain but only within the digital domain. It should not be possible for any actor to move across domains, but only in his current domain. That way programs executing on computers can move themselves to another computer and resume execution and actors in the physical domain can move to new locations. These constraints will be enforced by the semantics.

The **encrypt** and **decrypt** actions are special actions that an actor can perform to encrypt and decrypt data. Any data can be encrypted as long as the actor has access to the data. We do not allow already encrypted data to be encrypted again, and thus create a chain of encryption on the data for technical reasons. As described in the abstract system, $\mathcal{R}$ maps a location or data to its set of security restrictions, and to model the chain of encryption this mapping needs to be more flexible. The set of security restrictions would have to be a *list* of security restrictions where the ordering of restrictions would matter. We will not try to solve this problem in this version of the framework but see this as future work.

Decryption is only possible if the data allows the actor to perform the decryption, either through his location, his identity, or his keys. As mentioned before decryption is the only access restriction a datum can make, as data can be picked up and moved by any actor if the location gives the required access.

The second part of the syntax is shown in Figure 3.9. Formal fields are variables with an exclamation mark, and template fields can be formal fields, or tuple fields. The formal fields (written as $!x$ or $!u$) are used to bind variables to values. The formal fields can be used with the **in**, **read**, **encrypt**, and **decrypt** actions, and the use of them denotes that the actor "consumes" some data without knowing what it is. Tuple fields are locations, location variables, or expressions. Expressions can then be values or data variables.

ℓ	$::=$			LOCALITIES
			l	locality
		$\mid$	u	locality variable
N	$::=$			NETS
			$l ::^{\delta} [P]^{\langle n,\kappa\rangle}$	single node with a named process
		$\mid$	$l ::^{\delta} [\mathbf{nil}]$	anonymous node with an empty process
		$\mid$	$l ::^{\delta} \langle ef^{\rho}\rangle$	located tuple
		$\mid$	$N_1 \parallel N_2$	net composition
P	$::=$			PROCESSES
			$\mathbf{nil}$	null process
		$\mid$	$a.P$	action prefixing
		$\mid$	$P_1 \mid P_2$	parallel composition
		$\mid$	A	process invocation
a	$::=$			ACTIONS
			$\mathbf{out}(t)@\ell$	output
		$\mid$	$\mathbf{in}(T)@\ell$	input
		$\mid$	$\mathbf{read}(T)@\ell$	read
		$\mid$	$\mathbf{encrypt}(t,\rho,F)$	encryption
		$\mid$	$\mathbf{decrypt}(t,F)$	decryption
		$\mid$	$\mathbf{eval}(n,P)@l$	spawning of new programs
		$\mid$	$\mathbf{move}(l)$	movement of processes

Figure 3.8: Syntax of localities, nets, processes, and actions.

F	$::=$	$!x \mid !u$	formal fields	ef	$::=$	$V \mid l$	evaluated field
T	$::=$	$F \mid t$	template fields	e	$::=$	$V \mid x \mid \ldots$	expressions
t	$::=$	$e \mid \ell$	fields				

Figure 3.9: Syntax for template fields and fields.

3.8 Mapping Abstract Systems to insCalc

In this section we define how specifications in the specification language can be mapped to insCalc programs. The procedure is simple; we simply iterate over each location defined in the `locations` section of our specification and create a new process node in the resulting net. Each node gets the same name and access control attribute as in the specification language and an anonymous **nil** process. At the locations where an actors is present the process will be a process variable that is named with the name of the actor and has the actors keys available. There is no way to specify the spacial structure of a system with the insCalc syntax so we cannot map the connections of the specification language to anything useful. In a program implementing the insider framework the abstract system will be constructed by parsing the specification. Our running example expressed in the system specification language is given in Figure 3.10, and is mapped to a insCalc program as listed in Figure 3.11.

```
locations: HALL{*:m}(phys), JAN{key1:m}(phys),
        OFF{1234:m}(phys),
        SRV{4321:m}(phys), WASTE{SRV:i,r,o}(phys),
        PC1{PC2:e; pass:e,i,r,o}(dig),
        PC2{PC1:e; pass:e,i,r,o}(dig),
        PRT{PC1:o; PC2:o; SRV:i,r}(dig);
connections: HALL->JAN, JAN->HALL,
        HALL->OFF, OFF->HALL, HALL->SRV,
        SRV->HALL, OFF->PC1, SRV->PC2,
        SRV->WASTE, SRV->PRT, PC1->PC2,
        PC2->PC1, PC2->PRT, PC1->PRT;
actors: USER@OFF, JANITOR@JAN;
data: 1234{}@USER, key1{}@JAN, 4321{}@JAN,
      4321{}@USER, pass{}@USER;
```

Figure 3.10: The Running Example Modeled in the System Specification Language

$$
\begin{array}{ll}
\text{HALL} ::^{\langle * \to \mathsf{m} \rangle} [\mathbf{nil}] & || \\
\text{JAN} ::^{\langle \text{key1} \to \mathsf{m} \rangle} [\mathbf{J}]^{\langle \mathbf{JANITOR}, \{key1, 4321\} \rangle} & || \\
\text{OFF} ::^{\langle 1234 \to \mathsf{m} \rangle} [\mathbf{U}]^{\langle \mathbf{USER}, \{1234, 4321, pass\} \rangle} & || \\
\text{SRV} ::^{\langle 4321 \to \mathsf{m} \rangle} \mathbf{nil} & || \\
\text{WASTE} ::^{\langle \text{SRV} \to \mathsf{i,r,o} \rangle} [\mathbf{nil}] & || \\
\text{PC1} ::^{\langle \text{PC2} \to \mathsf{e}; \text{pass} \to \mathsf{e,i,r,o} \rangle} [\mathbf{nil}] & || \\
\text{PC2} ::^{\langle \text{PC1} \to \mathsf{e}; \text{pass} \to \mathsf{e,i,r,o} \rangle} [\mathbf{nil}] & || \\
\text{PRT} ::^{\langle \text{PC1} \to \mathsf{o}; \text{PC2} \to \mathsf{o}; \text{SRV} \to \mathsf{i,r} \rangle} [\mathbf{nil}] &
\end{array}
$$

Figure 3.11: The Running Example in insCalc

3.9 Semantics of insCalc

3.9.1 The Congruence Relation

The operational semantics of insCalc are given as a structural congruence on nets and processes and with a reduction relation over nets. The structural congruence $\equiv$ identifies, which nets intuitively represent the same net. The structural congruence relation which simplifies presentation of the semantics and the reasoning about processes is defined in Figure 3.12

$$
\begin{array}{ll}
\text{(Com)} & N_1 \mid\mid N_2 \equiv N_2 \mid\mid N_1 \\
\text{(Assoc)} & (N_1 \mid\mid N_2) \mid\mid N_3 \equiv N_1 \mid\mid (N_2 \mid\mid N_3) \\
\text{(Abs)} & l ::^{\delta} [P]^{\langle n,\kappa \rangle} \equiv l ::^{\delta} [(P \mid \mathbf{nil})]^{\langle n,\kappa \rangle} \\
\text{(Inv)} & l ::^{\delta} [A]^{\langle n,\kappa \rangle} \equiv l ::^{\delta} [P]^{\langle n,\kappa \rangle} \qquad \text{if } A \triangleq P \\
\text{(Clone)} & l ::^{\delta} [(P_1 \mid P_2)]^{\langle n,\kappa \rangle} \equiv l ::^{\delta} [P_1]^{\langle n,\kappa \rangle} \mid\mid l ::^{\delta} [P_2]^{\langle n,\kappa \rangle} \\
\text{(Anonym)} & l ::^{\delta} [\mathbf{nil}] \equiv l ::^{\delta} [\mathbf{nil}]^{\langle \epsilon,\emptyset \rangle}
\end{array}
$$

Figure 3.12: Structural Congruence on Nets and Processes.

The (Com) rule says that the order in which nodes in a net are presented does not matter. The net does not define any explicit spatial structure of the net, and thus the order of nodes in a net is irrelevant. The rule is the *commutative law* for nodes in a net. The (Assoc) rule says that $||$ is *associative*, that we can group nodes in anyway we want without creating a different net. The (Abs) says that we can safely append the **nil** process to any process P and not result in a different net. The (Inv) says that a process name A can be substituted out for the body of the process definition, if there exists a process definition for A. Thus an invocation of a process definition results in the body of the process definition getting substituted out for the process variable. The (Clone) rule says that a single process with two concurrent running parts at a single node can be split into two nodes, running each part separately. This will intuitively not create a different net, as the two process parts will be running at the same location as before, with the same access restrictions. The (Anonym) rule says that the anonymous **nil** process can be substituted out for a named **nil** process with an empty name and an empty set of keys. The rule basically says that the anonymous **nil** construct is "syntactic sugar" for a named **nil** process with an empty name and an empty set of keys.

3.9.2 The Reference Monitor

Access control is enforced by a *reference monitor* that is embedded in the operational semantics of the calculus. The reference monitor checks that access to data and localities is in accordance with the access policy of the data or locality. The reference monitor is defined in Figure 3.13 and Figure 3.14.

(1)

$$\text{grant} \;:\; \mathsf{Names} \times \mathsf{Loc} \times \mathcal{P}(\mathsf{Data}) \times \mathsf{AccMode} \times \mathsf{Loc} \times \mathsf{LocPolicy} \rightarrow \{\mathtt{true}, \mathtt{false}\}$$

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

(2)

$$\frac{l = l' \vee \exists (l, l') \in \mathsf{Con}}{\langle \mathcal{I}, n, \kappa \rangle \succ (l, l')}$$

(3)

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

Figure 3.13: Reference Monitor for Access Control, Part I

(4)

$$\text{decrypt} : \mathsf{Names} \times \mathsf{Loc} \times \mathcal{P}(\mathsf{Data}) \times \mathsf{Loc} \times \mathsf{Data} \rightarrow \mathcal{P}(\mathsf{Data})$$

$$\text{decrypt}(n, l, \kappa, et^{\rho}) = \begin{cases} \{et^{\emptyset}\} & \text{if } \rho = \emptyset \ \vee d \in \rho(n) \vee d \in \rho(l) \vee \\ & \exists k \in \kappa : d \in \rho(k) \vee \\ & \exists l'' \in \{l' \mid (l, l') \in \mathsf{Con} \wedge d \in \rho(l')\} \\ \emptyset & \text{otherwise} \end{cases}$$

(5)

$$\text{encrypt} : \mathsf{Names} \times \mathsf{Loc} \times \mathcal{P}(\mathsf{Data}) \times \mathsf{Data} \rightarrow \mathcal{P}(\mathsf{Data})$$

$$\text{encrypt}(n, l, \kappa, et^{\rho}, \rho') = \begin{cases} \{et^{\rho'}\} & \text{if decrypt}(n, l, \kappa, et^{\rho}) = \{et^{\emptyset}\} \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 3.14: Reference Monitor for Access Control, Part II

The predicate grant takes an actor n, a location l where the actor is located, a set of keys κ, the action a that the actor wants to perform, the location l' that the actor wants to perform the action on, and the access policy of that location as parameters. It only answers the question whether the actor has access to a specific location l' but does not take into account if the actor can actually reach the location l' from his current location l. The type of the a is $\mathsf{LocAccMode}$ as defined in Figure 3.3. The grant function is used to define the judgements $\leadsto$ and $\succ$.

The $\succ$ judgement specifies the conditions that must hold for an actor to be able to reach a location l' from l. The actor must be either at the location l' or in one of the neighboring location to l'. The $\leadsto$ judgement is then the reference monitor for location access, it specifies the conditions that need to hold for an actor n to be able to perform the action a in location l' when he is located at l. The conditions that must hold are that the actor is located in a neighboring location to l' and have access to perform the action a on the location l'. This distinguishes the reference monitor for insCalc from the one used in [16]. In the semantics in [16] the actors could magically perform actions on any location they could reach, and the movement of actors was implicit. We have decided to make the movement explicit, so an actor cannot perform an action on a location l unless he is located at the location or in a location that has an edge from the current location to l.

The decrypt function specifies the conditions that must hold for an actor n located at location l holding a set of keys κ to be able to decrypt the data et

which is encrypted by ρ. The function returns the decrypted data, i.e., the data without any access restrictions attached to it, if access is granted, and the empty set otherwise. The function decrypt is applied to data that is currently located at an actor, data at other actors or locations cannot be decrypted. The decrypt function checks if either the name of the actor, his location, one of the adjacent locations, or any one of his keys have access to the given data. It also checks if ρ is the empty set, as the empty security annotation means that the data is public.

The encrypt function specifies the conditions that must hold for an actor n located at l with keys κ to be able to encrypt the data et^{ρ} with a new key ρ'. As mentioned before we have decided that the actors can only encrypt public data, but not create a chain of encryptions on a datum. The encryption function thus checks to see if the actor is allowed to decrypt the data, and if he is able to do that the security annotation ρ is replaced with the new one ρ'. The function thus makes an implicit decrypt before it encrypts the data with a new key.

3.9.3 The Reduction Relation

The operational (small-step) semantics [12] for insCalc is presented in Figure 3.15 and Figure 3.16, and we now comment the semantic judgements. The judgements all have the form $\mathcal{S} \vdash N_1 \longmapsto \mathcal{S}' \vdash N_2$, where $\mathcal{S}$ is the abstract system presented in Section 3.4. The reference monitor part of each rule is located inside a box. Each judgement small-steps evaluates a net, and produces another net. For the **in**, **read**, **encrypt**, and **decrypt** actions only data that match the input templates are read. The matching is formalized in Figure 3.17. The match function produces a substitution that is used in the semantic judgements.

The (OUT) rule says that if an actor, located at l, performs the **out** action in location l' the datum will be added to the tuple space of l', if the following conditions hold:

- the location l' must exist in the abstract system,
- the security annotation of l' is δ',
- t evaluates to et^{ρ},
- the actor n has the required access to perform the **out** operation on the location l'. The $\rightsquigarrow$ relation ensures that the actor is either located at location l' or in a neighboring location.

(OUT)

$$\frac{l' \in \mathsf{Loc} \quad \mathcal{R}(l') = \delta' \quad [\![t]\!] = et^{\rho} \quad \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \mathsf{o}\rangle} \quad \mathcal{K}' = \mathcal{K}[l' \rightarrow \mathcal{K}[l'] \cup \{et^{\rho}\}]}{\mathcal{S} \vdash l ::^{\delta} [\mathbf{out}(t)@l'.P]^{\langle n,\kappa\rangle} \rightarrowtail \mathcal{S}' \vdash l ::^{\delta} [P]^{\langle n,\kappa\rangle} \mid\mid l' ::^{\delta'} \langle et^{\rho}\rangle}$$

(IN)

$$\frac{\begin{array}{c} match([\![T]\!], et^{\rho}) = \sigma \quad \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \mathsf{i}\rangle} \\ \mathcal{K}' = \mathcal{K}[l' \rightarrow \mathcal{K}[l'] \setminus et^{\rho}, n \rightarrow \mathcal{K}[n] \cup \{et^{\rho}\}] \end{array}}{\begin{array}{c} \mathcal{S} \vdash l ::^{\delta} [\mathbf{in}(T)@l'.P]^{\langle n,\kappa\rangle} \mid\mid l' ::^{\delta'} \langle et^{\rho}\rangle \rightarrowtail \\ \mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n,\kappa \cup \{et^{\rho}\}\rangle} \mid\mid l' ::^{\delta'} [\mathbf{nil}] \end{array}}$$

(READ)

$$\frac{match([\![T]\!], et^{\rho}) = \sigma \quad \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \mathsf{r}\rangle} \quad \mathcal{K}' = \mathcal{K}[n \rightarrow \mathcal{K}[n] \cup \{et^{\rho}\}]}{\begin{array}{c} \mathcal{S} \vdash l ::^{\delta} [\mathbf{read}(T)@l'.P]^{\langle n,\kappa\rangle} \mid\mid l' ::^{\delta'} \langle et^{\rho}\rangle \rightarrowtail \\ \mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n,\kappa \cup \{et^{\rho}\}\rangle} \mid\mid l' ::^{\delta'} \langle et^{\rho}\rangle \end{array}}$$

(ENCRYPT)

$$\frac{\mathrm{encrypt}(n, l, \kappa, [\![t]\!], \rho) = \{et^{\rho}\} \quad match([\![F]\!], et^{\rho}) = \sigma \quad \mathcal{R}' = \mathcal{R}[et \rightarrow \rho]}{\mathcal{S}' \vdash l ::^{\delta} [\mathbf{encrypt}(t, \rho, F).P]^{\langle n,\kappa\rangle} \rightarrowtail \mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n,\kappa \cup \{et^{\rho}\}\rangle}}$$

(DECRYPT)

$$\frac{\mathrm{decrypt}(n, l, \kappa, [\![t]\!]) = \{et^{\emptyset}\} \quad match([\![F]\!], et^{\emptyset}) = \sigma \quad \mathcal{R}' = \mathcal{R}[et \rightarrow \emptyset]}{\mathcal{S} \vdash l ::^{\delta} [\mathbf{decrypt}(t, F).P]^{\langle n,\kappa\rangle} \rightarrowtail \mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n,\kappa \cup et^{\emptyset}\rangle}}$$

Figure 3.15: Operational Semantics for insCalc, Part I

The rule uses the elements of $\mathcal{S}$ to verify the conditions and outputs a new datum node at l' if the conditions hold. The **out** action does not add a security annotation to the data, but assumes that the actor has used the **encrypt** action to specify the security restrictions. The $\mathcal{K}$ component of $\mathcal{S}$, which is the mapping of locations to data, is also updated by the rule in such a way that the data is added to the set of data at l'. The $\mathcal{S}'$ in the result reflects that the $\mathcal{K}$ component of $\mathcal{S}$ has been updated.

The (IN) says that an actor performing the **in** action reads and removes data from a location l'. The template T is used as a pattern to find which data at l' to retrieve. The *match* function creates a substitution, and the retrieved data will be substituted out for the reference to it in the resulting process. The actor will of course have to have access to perform the **in** action, and the rule uses the

$$\frac{l' \in \mathsf{Loc} \quad \mathcal{R}(l') = \delta' \quad \mathsf{Dom}(l) = \mathsf{Dom}(l') \quad \boxed{\langle \mathcal{S}, n, \kappa\rangle \rightsquigarrow \langle l, l', \mathsf{m}\rangle} \quad \mathcal{G}' = \mathcal{G}[n \to l']}{\mathcal{S} \vdash l ::^{\delta} [\mathbf{move}(l').P]^{\langle n,\kappa\rangle} \rightarrowtail \mathcal{S}' \vdash l ::^{\delta} [\mathbf{nil}] \mid\mid l' ::^{\delta'} [P]^{\langle n,\kappa\rangle}}\ (\textsc{move})$$

$$\frac{l' \in \mathsf{Loc} \quad \mathcal{R}(l') = \delta' \quad \mathsf{Dom}(l') = \mathit{digital} \quad \boxed{\langle \mathcal{I}, n, \kappa\rangle \rightsquigarrow \langle l, l', \mathsf{e}\rangle} \quad n' \notin \mathsf{Actors}}{\mathcal{S} \vdash l ::^{\delta} [\mathbf{eval}(n', Q)@l'.P]^{\langle n,\kappa\rangle} \rightarrowtail \mathcal{S}' \vdash l ::^{\delta} [P]^{\langle n,\kappa\rangle} \mid\mid l' ::^{\delta'} [Q]^{\langle n',\kappa\rangle}}\ (\textsc{eval})$$

$$\frac{\mathcal{S} \vdash N_1 \rightarrowtail \mathcal{S}' \vdash N_1'}{\mathcal{S} \vdash N_1 \mid\mid N_2 \rightarrowtail \mathcal{S}' \vdash N_1' \mid\mid N_2}\ (\textsc{par})$$

$$\frac{N \equiv N_1 \quad \mathcal{S} \vdash N_1 \rightarrowtail \mathcal{S}' \vdash N_2 \quad N_2 \equiv N'}{\mathcal{S} \vdash N \rightarrowtail \mathcal{S}' \vdash N'}\ (\textsc{struct})$$

Figure 3.16: Operational Semantics for insCalc, Part II

$$\text{match}(V, V) = \epsilon \qquad \text{match}(!x, V) = [V/x]$$
$$\text{match}(u, u) = \epsilon \qquad \text{match}(!u, \ell') = [\ell'/u]$$

Figure 3.17: Semantics for template matching.

reference monitor semantics to check that the actor is in a neighboring location, and that he has access to perform the **in** action on that location. When data is removed from the location l', the $\mathcal{K}$ component of $\mathcal{S}$ is updated to reflect that the data has moved, and is now with the given actor. As the data node at l' is removed from the net, an anonymous **nil** process is left at the location l' to ensure that the location does not disappear. This could happen if the data node was the only node defined for the location, and that would make it disappear from the net if we did not create a new node for it. Even though an actor has "consumed" data using the **in** rule, it does not mean that he can decrypt it and understand it. Although the user cannot understand the datum, it is added to his key set. When the user performs a successful **decrypt** action on the encrypted datum the decrypted datum, it will be available as a key to him and added to his key set κ.

The (READ) rule is the semantic rule for the **read** action. An actor performing the **read** action non-destructively reads data that is located at some location.

The match function creates a substitution matching some template T that the actor provides, and the substitution is applied on the remaining process. The actor n performing the **read** action on location l will of course have to have access to perform the **read** on l, and this condition is checked using the reference monitor semantics. If the template is successfully matched and the actor has the proper access, the datum et^{ρ} is added to the actor's key set, and the $\mathcal{K}$ component of $\mathcal{S}$ is updated. The datum is not removed from the location, so the location node is left unchanged in the resulting net. As with the (IN) rule, the actor is not guaranteed to be able to understand the read datum.

The **encrypt** action takes three arguments, a template field t, an access policy ρ, and a formal F. The action will try to put the access policy ρ on t and return the result to F. The (ENCRYPT) rule uses the encrypt function from the reference monitor semantics to perform the encryption, and if it is successful a substitution is created and applied to the remaining process. The $\mathcal{R}$ part of $\mathcal{S}$, which is the mapping of data to its access policy, is updated to the new access policy ρ and the encrypted datum is added to the actor's key set. The reference monitor semantics will ensure that only data, which the actor has access to, can be encrypted, so the actor cannot "steal" data that is not intended for him. The actor will neither be able to encrypt already encrypted data, the policy on the data will be substituted for a new policy if the actor has the proper access.

The (DECRYPT) rule describes the semantics for the **decrypt** action. An actor performing the **decrypt** action will only succeed if he has the necessary access to the datum. Decryption of data will strip of all access restriction on the data which will be added to the actor's key set and become usable as a key. If the decrypt function from the reference monitor semantics is successful, it will return the datum without any security policy. A substitution will be generated that substitutes all references to the decrypted data with its value in the resulting process. The $\mathcal{R}$ component of $\mathcal{S}$ is updated in such a way that the policy for et is the empty set.

For the actions **move** and **eval** rules there are two integrity checks that the semantics must enforce when an actor is performing either **move** or **eval**. The first check is that an actor can only **move** in one domain, never across domain borders, as this would enable "human" actors to become programs and computer programs to become human. Both actors in the digital and the physical domain are only allowed to move to another location within their domain. The second check is that a new process/actor that is spawned by the **eval** action must be located at the digital domain. Both computers and human actors can spawn new processes, but all new processes must run on computers.

The (MOVE) rule describes how processes can move between locations in one step. We have defined the reference monitor in such a way that actions can only be done on the current location of an actor or an adjacent location. This also applies for the **move** action. For an actor n to be able to move to location l' the following conditions must hold:

- the location l' must exist in the set of nodes Loc,
- the access policy for l' is δ',
- the current domain of the actor l must be the same as the domain of l',
- the actor must have access to move to l' and be able to do so in one step. The reference monitor rule $\rightsquigarrow$ will take care of these two conditions.

If all of these conditions hold, the $\mathcal{G}$ component of S, which is the mapping of actors to locations, is updated to reflect that n is moved to l'. In the resulting net an anonymous **nil** process is left at location l to ensure that the location does not disappear from the net, as the process that moved could have been the only element at the location. The process that moved will then continue evaluation at l'.

The (EVAL) rule describes how to spawn new processes in the digital domain. We have decided that new processes can only be created in the digital domain to model the execution of programs. Actors in the digital domain and in the physical domain can spawn new processes but the new location of the new process has to be the digital domain. The following conditions must hold in order for a process to be able to start a new process at a location l'

- the location l' must exist in the set of nodes Loc,
- the access policy for l' is δ',
- the domain of the location the new process should run at l' must be *digital*
- the actor must have access to perform the **eval** action on the given location l' and must be able to reach the location. The reference monitor semantics $\rightsquigarrow$ handles these two conditions.

If these conditions hold the new process will be added to the set of actors Actors and be given a unique identifier. It is not possible to add an actor with the same name twice. The new process will have the same key set κ as the process that created it.

The (PAR) rule says that a semantic evaluation of a single node in the net will result in the whole net being updated. This is the starting rule for a semantic evaluation where a scheduler is free to choose any node in the net and evaluate it.

The last rule is (STRUCT). It relates the semantic reduction rules to the congruence relation. The rule says that it is always safe to use a congruence rule to change a single node, without changing the semantics of the whole net.

3.10 The Analysis

This section describes two insider analyses that we have defined for the framework described so far. The analyses are the first of three analyses that we define for the insider framework. For the third analysis, the framework will have to be extended to enable logging of actions.

3.10.1 Reachability analysis (Analysis0)

The first analysis is defined on the spatial structure of the system and is a graph based analysis. The idea is to calculate an over-approximation of the set of data and locations the actors in the system can reach or access. The result of the analysis should be the set of locations and the set of data each actor can reach, i.e. $locs : \mathsf{Actors} \to \mathcal{P}(\mathsf{Loc})$ and $data : \mathsf{Actors} \to \mathcal{P}(\mathsf{Data})$. The analysis can be viewed as a "before-the-fact" analysis and used to ensure that the designed system does not have any serious security flaws.

Finding an over-approximation of the set of locations a given actor can reach is not as simple as it sounds. We could, of course, just define the sets as all the locations and all the data in the system, but that would not be satisfactory, as some locations would never become available to the actor along any path whatever actions the actors might take. The challenge is that as an actor roams a system he can pick up data, which can give him access to new locations and so on. If we were extremely paranoid we would also take into account actions performed by other actors in the system. An actor could leave a datum at a location where another actor could pick it up and use as a key to reach another location. In our analysis we do not take into account actions performed by other actors, and the only data that matters to our analysis is the data that the actor has available initially and the data lying at locations. Data at other actors will be ignored. In later analyses we will take into account actions performed by other actors, but it does not seem to be suitable in a "before-the-fact" analysis.

3.10.1.1 Algorithm for Analysis0

The analysis focuses on a single given actor and solves the problem for one actor at a time. The analysis begins finding all reachable locations, for a given actor, starting from his initial location. The algorithm uses the *grant* function as defined in Figure 3.13 to check which location the actor can move to from a given location. Our approach to solving the problem is to place the actor in his initial location, recursively calculate to which locations he can move, and make the actor read and decrypt all data that he finds along the way. Also the actor must try to decrypt all encrypted data in his key set at each location, as he might pick up a key or be located in a location that makes it possible to decrypt a datum that is encrypted in his key set.

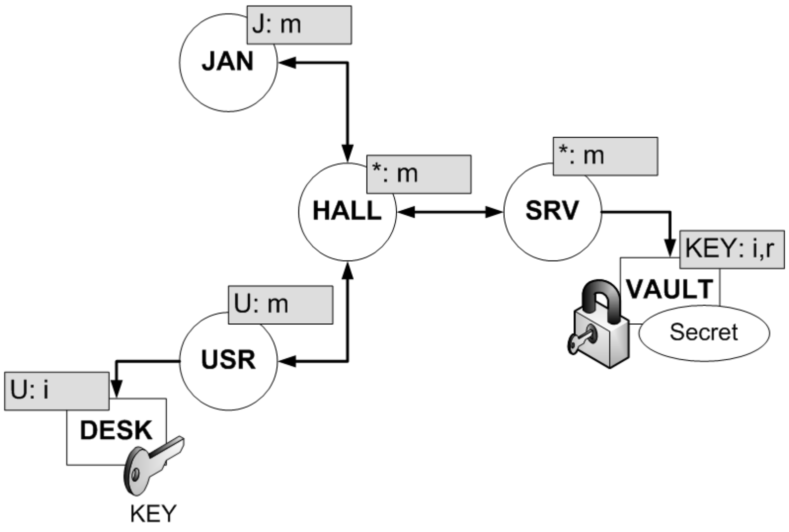


Figure 3.18: The running example as a graph

Because the actor can use data that he picks up as keys, the order in which he visits the locations is important. We will thus have to repeat this process, each time with a bigger key set, until his key set does not change anymore, i.e., κ reaches a *fixed-point*. By repeating the process, the algorithm is simulating every possible path that the actor could have taken and thus results in a superset of the locations and data the actor can reach.

In the example presented in Figure 3.18 the algorithm should be able to calculate that only the actor U can get the "Secret" data that is stored in the vault.

The reachability analysis is defined by the algorithms listed in Algorithm 4 to

Algorithm 6. *analysis0* is the main loop of the algorithm, it places the actor in his initial location and initializes the result sets, before calling the findLoc function. The *findLoc* function is in charge of re-running the analysis for a given actor at a given location, until the key set and location set reach a fixed point. The *checkLoc* function is in charge of collecting information about where the actor has been and which data he has possibly collected. It is a recursive function and it keeps track of locations it has already visited to avoid being stuck in a loop going back and forth between two location.

Algorithm 4 getLoc($\mathsf{Names} \times \mathsf{Loc} \times \mathsf{Keys} \times \mathcal{P}(\mathsf{Loc}) \times \mathsf{System}) \to (\mathcal{P}(\mathsf{Loc}), \mathcal{P}(\mathsf{Data}))$

```
checkLoc($n, l, \kappa, locs, \mathcal{S}$) =
$\kappa'$ := decryptAll($n, l, \kappa$)
$locs' := locs \cup \{l\}$
for all $(l, l') \in \mathsf{Con} : l' \notin locs'$ do
  if $\langle \mathcal{S}, n, \kappa' \rangle \rightsquigarrow (l, l', \mathsf{i}) \vee \langle \mathcal{S}, n, \kappa' \rangle \rightsquigarrow (l, l', \mathsf{r})$ then
    $\kappa' := \kappa' \cup \mathcal{K}(l')$ {get all data at $l'$ where $n$ has "input" access}
    $locs' := locs' \cup \{l'\}$
  else if $\langle \mathcal{S}, n, \kappa' \rangle \rightsquigarrow (l, l', \mathsf{m})$ then
    $(locs', \kappa'')$ := checkLoc($n, l', \kappa', locs', \mathcal{S}$) {If the actor can reach another
      location recursively continue}
    $locs' := locs' \cup \{l'\}$
    $\kappa' := \kappa''$
  end if
end for
return $(locs', \kappa')$
```

```
decryptAll($n, l, \kappa$) =
$\kappa' := \emptyset$
for all $k \in \kappa$ do
  $\kappa' = \kappa' \cup$ decrypt($n, l, \kappa, k$)
end for
if $\kappa' \neq \emptyset$ then
  return $\kappa' \cup$ decryptAll($n, l, a, \kappa \cup \kappa'$)
end if
return $\kappa'$
```

3.10.1.2 Running analysis0

We now run analysis0 on the system in Figure 3.18 for actor U. The first two iterations of the algorithm processes are shown in Figure 3.19. This graph shows that after the first two iterations the result for this user is already computed.

Algorithm 5 findLoc(Names × Loc × Keys × System) → ($\mathcal{P}$(Loc), $\mathcal{P}$(Data))

```
1: findLoc($n, l, \kappa, \mathcal{S}$) =
2: ($locations, \kappa$') := checkLoc($n, l, \kappa, \{\}, \mathcal{S}$)
3: if $\kappa'$ != $\kappa$ then
4:    findLoc($n, l, \kappa', \mathcal{S}$)
5: else
6:    return ($locations, \kappa'$)
7: end if
```

Algorithm 6 analysis0(Names × Loc × Keys × System) → ($\mathcal{P}$(Loc), $\mathcal{P}$(Data))

```
1: analysis0($n, l, \kappa, \mathcal{S}$) =
2: ($locs, data$) := findLoc($n, l, \kappa, \mathcal{S}$)
3: return ($locs, data$)
```

The algorithm will continue with the other locations, of course, but no additional data will be added to the result sets.

3.10.2 Control Flow Analysis (Analysis1)

The reachability analysis does not take into account what actions a given actor actually performs in the system, and is only intended to be a "before-the-fact" analysis. The second analysis we present takes into account the actions performed by all actors in the system. The analysis will work directly on insCalc programs in which we will provide a process definition for the actor's process variable. The process definition can be viewed as a sequence of actions reconstructed from a log file after an incident has occurred. We assume that every single action performed by an actor is recorded so the analysis will not have to "guess" any actions. We assume that the process definition will be made available to our analysis, and it will be substituted out for the process variable in the insCalc program. We do not allow the process definition to call other process variables in the process definitions, so there is no way of calling a process definition recursively. The reason for this is that there is no means of storing the process definitions in the insCalc program and thus we can only make "inline" substitutions of process variables.

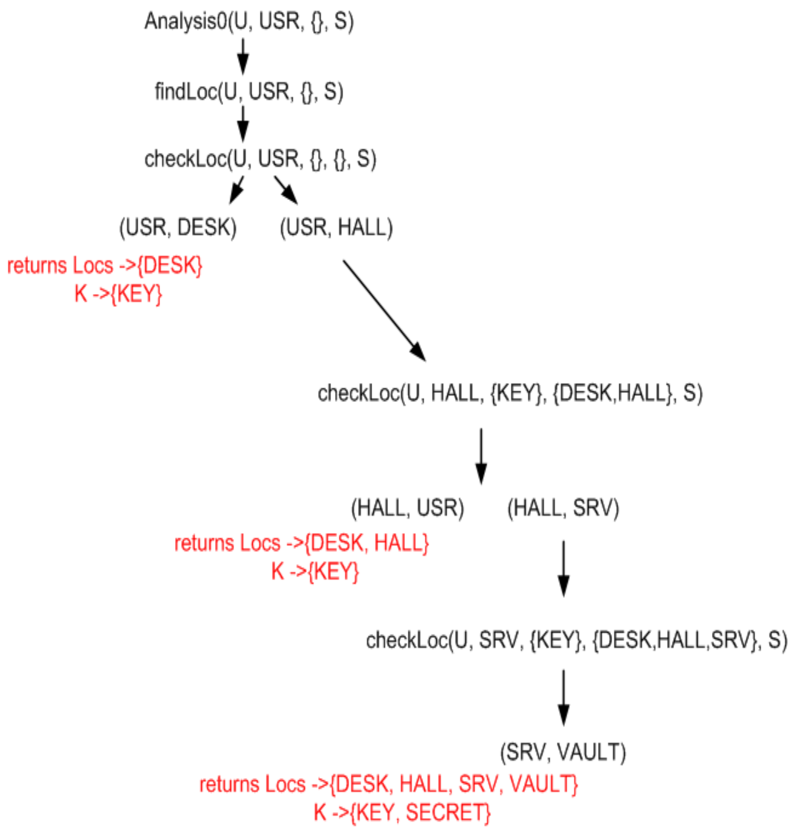

Analysis0(U, USR, {}, S)
findLoc(U, USR, {}, S)
checkLoc(U, USR, {}, {}, S)
(USR, DESK)
(USR, HALL)
returns Locs ->{DESK}
K ->{KEY}
checkLoc(U, HALL, {KEY}, {DESK,HALL}, S)
(HALL, USR)
(HALL, SRV)
returns Locs ->{DESK, HALL}
K ->{KEY}
checkLoc(U, SRV, {KEY}, {DESK,HALL,SRV}, S)
(SRV, VAULT)
returns Locs ->{DESK, HALL, SRV, VAULT}
K ->{KEY, SECRET}

Figure 3.19: The running example as a graph

3.10.2.1 Flow Logic Specification

We begin by giving a Flow Logic specification for the analysis, which we use to implement an algorithm to compute an analysis estimate. We do not use the specification in the traditional way, i.e., to create a set of constraints that are solved by a solver. We consider this to be future work. Instead we will describe an algorithm that is guided by both the semantics of insCalc and the Flow Logic specification, and computes an analysis estimate. The analysis estimate for the analysis should be a the set of data that a given actor can obtain by executing the given sequence of actions and the set of locations that he can reach by performing the given sequence of actions.

The Flow Logic specification is given as a number of judgements, one for each

$$
\begin{array}{lll}
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{N}} l ::^{\delta} [P]^{\langle n,\kappa\rangle} & \text{iff} & (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} P \\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{N}} l ::^{\delta} [\mathbf{nil}] & \text{iff} & (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} \mathbf{nil} \\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{N}} l ::^{\delta} \langle et\rangle & \text{iff} & et \in \hat{T}(l) \\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{N}} N_1 \mid\mid N_2 & \text{iff} & (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{N}} N_1 \wedge (\hat{T},\hat{\sigma},\mathcal{I}) \models_{\mathrm{N}} N_2 \\
\\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} \mathbf{nil} & \text{iff} & \mathit{true} \\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} P_1 \mid P_2 & \text{iff} & (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} P_1 \wedge (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} P_2 \\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} A & \text{iff} & (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} P \quad \text{if } A \triangleq P \\
\\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} \mathbf{out}(t)@\ell.P & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell) : (\langle \mathcal{S},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathsf{o}) \Rightarrow \\
 & & \hat{\sigma}[\![t]\!] \subseteq \hat{T}(\hat{l})) \wedge (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} P \\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} \mathbf{in}(T)@\ell.P & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell) : (\langle \mathcal{S},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathsf{i}) \Rightarrow \\
 & & \hat{T}(\hat{l}) \subseteq \hat{\sigma}(\ell) \wedge (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} P \\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} \mathbf{read}(T)@\ell.P & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell) : (\langle \mathcal{S},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathsf{r}) \Rightarrow \\
 & & \hat{T}(\hat{l}) \subseteq \hat{\sigma}(\ell)) \wedge (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} P \\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} \mathbf{eval}(n',Q)@\ell.P & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell) : (\langle \mathcal{S},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathsf{e}) \Rightarrow \\
 & & \wedge(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{p}}^{\hat{l},n',\kappa} Q)(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} P \\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} \mathbf{encrypt}(t,\rho,F) & \text{iff} & ((\mathit{enc} = \mathrm{encrypt}(n,l,\kappa,t,\rho) \wedge \mathit{enc} \neq \emptyset) \\
 & & \Rightarrow \hat{\sigma} \models_1 F : \hat{T}(\hat{l}) \rhd \hat{W}\bullet) \wedge \\
 & & (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} P \\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} \mathbf{decrypt}(t,F).P & \text{iff} & ((\mathit{dec} = \mathrm{decrypt}(n,l,\kappa,t) \wedge \mathit{dec} \neq \emptyset) \\
 & & \Rightarrow \hat{\sigma} \models_1 F : \hat{T}(\hat{l}) \rhd \hat{W}\bullet) \wedge \\
 & & (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} P \\
(\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{l,n,\kappa} \mathbf{move}@\ell.P & \text{iff} & \forall \hat{l} \in \hat{\sigma}(\ell') : (\langle \mathcal{S},n,\kappa\rangle \rightsquigarrow (l,\hat{l},\mathsf{m}) \Rightarrow \\
 & & (\hat{T},\hat{\sigma},\mathcal{S}) \models_{\mathrm{P}}^{\hat{l},n,\kappa} P)
\end{array}
$$

Figure 3.20: Flow Logic Specification for Analysis1.

syntactic category of the insCalc language. For each syntactic construct there is one inference rule that describes the conditions that must hold for an analysis estimate to be correct. The information is collected in two components: $\hat{T}$ and $\hat{\sigma}$. The $\hat{T}$ records for every tuple space the set of tuples possibly located in that tuple space at any given point in time. $\hat{\sigma}$ tracks possible values that variables may be bound to during execution. The Flow Logic specification is listed in Figure 3.20 and Figure 3.21.

The clauses listed in Figure 3.20 define two judgements : $\models_{\mathrm{N}}$ and $\models_{\mathrm{P}}$. The clauses are syntax-directed and make decisions about the next step of evaluation on the syntactical constructs it meets. The $\models_{\mathrm{N}}$ judgement defines the rules for processing nets, and $\models_P$ defines the rules for processing processes. In order to be able to keep the processing of nets going we need the sets $\hat{T}$ and $\hat{\sigma}$ to keep track of data at locations and data at variables in the net.

$$\begin{array}{lll}
\sigma \models_1 V : \hat{V}_\circ \rhd \hat{W}_\bullet & \text{iff} & \{e \in \hat{V}_\circ \mid e = V\} \sqsubseteq \hat{W}_\bullet \\
\sigma \models_1 l : \hat{V}_\circ \rhd \hat{W}_\bullet & \text{iff} & \{e \in \hat{V}_\circ \mid e = l\} \sqsubseteq \hat{W}_\bullet \\
\sigma \models_1 x : \hat{V}_\circ \rhd \hat{W}_\bullet & \text{iff} & \{e \in \hat{V}_\circ \mid e \in \sigma(x)\} \sqsubseteq \hat{W}_\bullet \\
\sigma \models_1 u : \hat{V}_\circ \rhd \hat{W}_\bullet & \text{iff} & \{e \in \hat{V}_\circ \mid e \in \sigma(u)\} \sqsubseteq \hat{W}_\bullet \\
\sigma \models_1 !x : \hat{V}_\circ \rhd \hat{W}_\bullet & \text{iff} & \hat{V}_\circ \sqsubseteq \sigma(x) \\
\sigma \models_1 !u : \hat{V}_\circ \rhd \hat{W}_\bullet & \text{iff} & \hat{V}_\circ \sqsubseteq \sigma(u)
\end{array}$$

Figure 3.21: Flow Logic Specification for Analysis1, Part II.

The clauses in the Figure 3.20 define the judgement $\models_1$ for analyzing pattern matching. The general form of a rule is $\sigma \models_1$ *pattern* : $\hat{V}_\circ \rhd \hat{W}_\bullet$, where σ is the variable environment, *pattern* is the pattern to be matched, $\hat{V}_\circ$ is the tuple space where the pattern should be matched, and finally $\hat{W}_\bullet$ is the result of the matching. The first two rules say that if the pattern to be matched is data or a location, the value must be present in the tuple space. The next two rules say that if the pattern is a data variable or a location variable the result will be all the values the variable can take which also are in the tuple space. The final rules say that if the pattern is a formal field, a new variable, the variable is added to σ and it is given all the values in the tuple space.

3.10.2.2 Language for Specifying Process Definitions

As we are interested in an implementable insider framework we must be able to provide the process definition for all process variables in the generated insCalc program, so we need a language for specifying the process sequence. Figure 3.22 and Figure 3.23 describe the syntax of the language in which the user of our analysis tool can specify the process definition. There must be a definition for each process variable defined in the system and there is no way of calling other process definitions. The syntax is closely related to insCalc syntax the syntax of patterns, and is basically a list of definition of process variables.

3.10.2.3 Algorithm for Calculating the Analysis Estimate

The algorithm we now describe uses the rules in the Flow Logic specification to calculate an estimate for the data in $\hat{T}$ and $\hat{\sigma}$. It also calculates an estimate of the data in each actor's key set and an estimate of the locations visited by an

$DefList$	$::=$	Def	a single definition
	$\mid$	$Def; DefList$	a list of definition
Def	$::=$	$\mathsf{Names} := ProcessExpr$	a definition
$ProcessExpr$	$::=$	nil	the nil process
	$\mid$	$Action.ProcessExpr$	an action
	$\mid$	$ProcessExpr \mid ProcessExpr$	concurrent process
$Action$	$::=$	out$(Field)@Locality,$	the out action
	$\mid$	in$(Template)@Locality$	the in action
	$\mid$	read$(Template)@Locality$	the read action
	$\mid$	encrypt$(Field, DataAccess,$ $Formal)$	the encrypt action
	$\mid$	decrypt$(Field, Formal)$	the decrypt action
	$\mid$	eval$(NAME, ProcessExpr)@NAME$	the eval action
	$\mid$	move$(NAME)$	the move action

Figure 3.22: Language Specification for Process Definitions, Part I

$Locality$	$:=$	"Names"	locality value
	$\mid$	Names	locality variable
$Formal$	$:=$	$!\mathsf{Names}$	formals
$Template$	$:=$	$Formal$	formals
	$\mid$	$Field$	field
$Field$	$:=$	$Locality$	locality
	$\mid$	$Expr$	variable
$Expr$	$:=$	"Names"	data value
	$\mid$	Names	data variable
$DataAccess$	$::=$	ϵ	empty policy list
	$\mid$	$DataPolicy$	a single policy
	$\mid$	$DataPolicy; DataAccess$	a data policy list
$DataPolicy:$	$::=$	$\mathsf{Names} : DataAccMode$	
	$\mid$	$* : DataAccMode$	
$DataAccMode$	$::=$	ϵ	empty access
	$\mid$	$\mathtt{d}$	decrypt

Figure 3.23: Language Specification for Process Definitions, Part II

actor into the sets $\hat{\kappa}$ and $\hat{L}$ respectively. We are interested in a *static analysis* that calculates the estimates, and cannot *run* the semantics of the program in any way - only simulate the result. We must only use the semantics to guide the collection of data into the result sets. The idea is to use the rules from the Flow Logic specification to define an algorithm that processes insCalc programs in witch the process variables have been substituted for process definitions.

The algorithm starts processing the net, one node at a time, and for each language construct applies the appropriate rules similar to the ones in the Flow Logic specification. At each rule, instead of checking for membership of data in $\hat{T}$ and $\hat{\sigma}$, the algorithm adds data to the sets if the proper conditions are fulfilled. To model all possible interleavings of actions taken by all actors, we repeat the processing of the entire net until $\hat{L}$, $\hat{\kappa}$, $\hat{T}$, $\hat{\sigma}$ reach a fixed-point. Each time an actor reads data, takes data from a location, decrypts or encrypts data, his keys set grows. The algorithm is listed in Algorithm 7 to Algorithm 11. The function *analysis1* is the main loop, and repeats the processing of the net until the four sets reach a fixed point. The function N is responsible for processing the nodes in the net, and there is a rule for each possible type of node. The function P is responsible for processing a process and recursively processes one action at a time. The processing of nets closely follows the Flow Logic specification. The only thing that could seem strange is that when an actor **inputs** data, i.e., removes data from a tuple space, the data is not removed in the algorithm. The reason for this is that we do not know in which order the actors are reading and outputting data in the system, and we cannot remove anything as some actor could be able to read the data before the data was removed by another.

Algorithm 7 $\mathrm{N}((\mathsf{Loc}\times\mathcal{P}(\mathsf{Data}))\times(\mathsf{Names}\times\mathcal{P}(\kappa))\times(\mathsf{Actors}\times\mathcal{P}(\mathsf{Loc}))\times(\mathsf{Actors}\times\mathcal{P}(\mathsf{Data}))\times\mathsf{Net}\times\mathsf{System})\rightarrow((\mathsf{Loc}\times\mathcal{P}(\mathsf{Data}))\times(\mathsf{Names}\times\mathcal{P}(\mathsf{Data}))\times(\mathsf{Actors}\times\mathcal{P}(\mathsf{Loc}))\times(\mathsf{Actors}\times\mathcal{P}(\mathsf{Data})))$

```
N($\hat{T}, \hat{\sigma}, \hat{\kappa}, \hat{L}, Net, \mathcal{S}$) =
if $net = \ell ::^{\delta} [P]^{\langle n,\kappa\rangle}$ then
    $\hat{\kappa}(n) := \hat{\kappa}(n) \cup \kappa$
    $\hat{L}(n) := \hat{L}(n) \cup \{l\}$
    return P($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, n, P$)
else if $net = \ell ::^{\delta} [\mathbf{nil}]^{\langle n,\kappa\rangle}$ then
    return ($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}$)
else if $net = \ell ::^{\delta} \langle et^{\rho}\rangle$ then
    return ($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}$)
else if $net = \ell ::^{\delta} N_1 \mid\mid N_2$ then
    ($\hat{T}', \hat{\sigma}', \hat{L}', \hat{\kappa}'$) :=N($\hat{T}, \hat{\sigma}, \hat{\kappa}, \hat{L}, N_1$)
    return N($\hat{T}', \hat{\sigma}', \hat{\kappa}', \hat{L}', N_2$)
end if
```

Algorithm 8 $\mathrm{P}((\mathsf{Loc} \times \mathcal{P}(\mathsf{Data})) \times (\mathsf{Names} \times \mathcal{P}(\mathsf{Data})) \times (\mathsf{Actors} \times \mathcal{P}(\mathsf{Loc})) \times (\mathsf{Actors} \times \mathcal{P}(\mathsf{Data})) \times \mathcal{P}(\mathsf{Data}) \times \mathsf{Loc} \times \mathsf{Actors} \times \mathsf{Proc} \times \mathsf{System}) \to ((\mathsf{Loc} \times \mathcal{P}(\mathsf{Data})) \times (\mathsf{Names} \times \mathcal{P}(\mathsf{Data})) \times (\mathsf{Actors} \times \mathcal{P}(\mathsf{Loc})) \times (\mathsf{Actors} \times \mathcal{P}(\mathsf{Data})))$

```
1: P($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n, proc, \mathcal{S}$) =
2: if $proc$ = nil then
3:     return $(\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa})$
4: else if $proc = P_1 \mid P_2$ then
5:     $(\hat{T}', \hat{\sigma}', \hat{L}', \hat{\kappa}') := \mathrm{P}(\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n, P_1)$
6:     return $\mathrm{P}(\hat{T}', \hat{\sigma}', \hat{L}', \kappa', \ell, n, P_2)$
7: else if $proc = \mathbf{out}(t)@\ell.P$ then
8:     for all $\hat{l} \in \hat{\sigma}(\ell)$ do
9:         if $(\langle \mathcal{S}, n, \hat{\kappa}(n) \rangle \rightsquigarrow (l, \hat{l}, \mathsf{o}))$ then
10:            $\hat{T}(\hat{l}) := \hat{\sigma}(t)$
11:            $\hat{L} := \hat{L} \cup \{\hat{l}\}$
12:        end if
13:    end for
14:    return $\mathrm{P}(\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n, P)$
15: else if $proc = \mathbf{in}(T)@\ell.P$ then
16:    for all $\hat{l} \in \hat{\sigma}(\ell)$ do
17:        if $(\langle \mathcal{S}, n, \hat{\kappa}(n) \rangle \rightsquigarrow (l, \hat{l}, \mathsf{i}))$ then
18:            if T = !x or T = !u then
19:                $\hat{\sigma}(T) := \hat{\sigma}(T) \cup \hat{T}(l)$
20:                $\hat{\kappa}(n) := \hat{\kappa}(n) \cup \hat{T}(l)$
21:            else if T = x or T = u then
22:                $\hat{\kappa}(n) := \hat{\kappa}(n) \cup \hat{\sigma}(T)$
23:            else if T = V or T = l then
24:                $\hat{\kappa}(n) := \hat{\kappa}(n) \cup \{T\}$
25:            end if
26:            $\hat{L}(n) := \hat{L}(n) \cup \{\hat{l}\}$
27:        end if
28:    end for
29:    return $\mathrm{P}(\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n, P)$
30: end if
```

3.10.3 Running Analysis1

We will now run the analysis for a single process in the example presented in Figure 3.18. We place the actor U in the USR location and place the process for the actor in the USR location. The process is a simple sequence of actions where the actor takes the key from his desk, moves to the server room, opens the vault, reads the secret and returns to his desk. The insCalc net for this

Algorithm 9 P((Loc × $\mathcal{P}$(Data)) × (Names × $\mathcal{P}$(Data)) × (Actors × $\mathcal{P}$(Loc)) × (Actors×$\mathcal{P}$(Data))×$\mathcal{P}$(Data)×Loc×Actors×Proc×System) → ((Loc×$\mathcal{P}$(Data))× (Names × $\mathcal{P}$(Data)) × (Actors × $\mathcal{P}$(Loc)) × (Actors × $\mathcal{P}$(Data)))) - Continued I

```
P($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n, proc, \mathcal{S}$) =
if $proc = \mathbf{read}(T)@\ell.P$ then
    for all $\hat{l} \in \hat{\sigma}(\ell)$ do
        if $(\langle \mathcal{S}, n, \hat{\kappa}(n)\rangle \rightsquigarrow (l, \hat{l}, \mathsf{r}))$ then
            if T = !x or T = !u then
                $\hat{\sigma}(T) := \hat{\sigma}(T) \cup \hat{T}(l)$
                $\hat{\kappa}(n) := \hat{\kappa}(n) \cup \hat{T}(l)$
            else if T = x or T = u then
                $\hat{\kappa}(n) := \hat{\kappa}(n) \cup \hat{\sigma}(T)$
            else if T = V or T = l then
                $\hat{\kappa}(n) := \hat{\kappa}(n) \cup \{T\}$
            end if
            $\hat{L}(n) := \hat{L}(n) \cup \{\hat{l}\}$
        end if
    end for
    return P($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n, P$)
else if $proc = \mathbf{eval}(n', T)@\ell.P$ then
    for all $\hat{l} \in \hat{\sigma}(\ell)$ do
        if $(\langle \mathcal{S}, n, \hat{\kappa}(n)\rangle \rightsquigarrow (l, \hat{l}, \mathsf{e}))$ then
            $\hat{L}(n) := \hat{L}(n) \cup \{\hat{l}\}$
            $((\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa})) :=$ P($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n', Q$)
        end if
    end for
    return P($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n, P$)
end if
```

system will look like the on in Figure 3.25. The algorithm for analysis1 will process each node in the order it is presented in the net. Before the algorithm runs, the abstract system $\mathcal{S}$ must be established as the algorithm uses that to find data at location and to check for access rights. The algorithm starts by processing the node for the HALL location by splitting the net into the HALL node and the rest of the net. Nothing interesting happens until the algorithm starts processing the USR node which contains the process for the actor in the system. Figure 3.24 shows the analysis result of processing each action. This simple example is straight forward but would quickly become more complicated with more actors and more locations. We shall run more interesting systems in the chapter on evaluation.

Algorithm 10 P((Loc $\times$ $\mathcal{P}$(Data)) $\times$ (Names $\times$ $\mathcal{P}$(Data)) $\times$ (Actors $\times$ $\mathcal{P}$(Loc)) $\times$ (Actors$\times\mathcal{P}$(Data))$\times\mathcal{P}$(Data)$\times$Loc$\times$Actors$\times$Proc$\times$System) $\rightarrow$ ((Loc$\times\mathcal{P}$(Data))$\times$ (Names $\times$ $\mathcal{P}$(Data)) $\times$ (Actors $\times$ $\mathcal{P}$(Loc)) $\times$ (Actors $\times$ $\mathcal{P}$(Data)))) - Continued II

```
P($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n, proc, \mathcal{S}$) =
if $proc$ = **encrypt**$(t, \rho, T).P$ then
    enc := $encrypt(n, l, \hat{\kappa}(n), t, \rho)$
    if enc != $\emptyset$ then
        if T = !x or T = !u then
            $\hat{\sigma}(T) := \hat{\sigma}(T) \cup enc$
        end if
    end if
    return P($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n, P$)
else if $proc$ = **decrypt**$(t, \rho, T).P$ then
    dec := $decrypt(n, l, \hat{\kappa}(n), t)$
    if enc != $\emptyset$ then
        if T = !x $\vee$ T = !u then
            $\hat{\sigma}(T) := \hat{\sigma}(T) \cup dec$
        end if
    end if
    return P($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n, P$)
else if $proc$ = **move**$@\ell.P$ then
    for all $\hat{l} \in \hat{\sigma}(\ell)$ do
        if $(\langle \mathcal{S}, n, \hat{\kappa}(n) \rangle \rightsquigarrow (l, \hat{l}, \mathsf{m}))$ then
            $\hat{L}(n) := \hat{L}(n) \cup \{\hat{l}\}$
        end if
    end for
    return P($\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, l, n, P$)
end if
```

Algorithm 11 analysis1

```
analysis1($net, \mathcal{S}$) =
$\hat{L}$;$\hat{L}'$;$\hat{\kappa}$;$\hat{\kappa}'$;$\hat{T}$;$\hat{T}'$;$\hat{\sigma}$;$\hat{\sigma}$' := {}
repeat
    $\hat{L} := \hat{L}'$
    $\hat{\kappa} := \hat{\kappa}'$
    $\hat{T} := \hat{t}'$
    $\hat{\sigma} := \hat{\sigma}'$
    $(\hat{T}', \hat{\sigma}', \hat{L}', \hat{\kappa}') := N(\hat{T}, \hat{\sigma}, \hat{L}, \hat{\kappa}, net)$
until $\hat{L}' = \hat{L} \wedge \hat{\kappa} = \hat{\kappa}' \wedge \hat{T}' = \hat{T} \wedge \hat{\sigma}' = \hat{\sigma}$

```

Action	$\hat{\sigma}$	$\hat{\kappa}$	$\hat{L}$
input(!key) @DESK	$\{key \rightarrow \{KEY\}\}$	$\{U \rightarrow \{KEY\}\}$	$\{U \rightarrow \{USR, DESK\}\}$
move.HALL	$\{key \rightarrow \{KEY\}\}$	$\{U \rightarrow \{KEY\}\}$	$\{U \rightarrow \{USR, DESK, HALL\}\}$
move.SRV	$\{key \rightarrow \{KEY\}\}$	$\{U \rightarrow \{KEY\}\}$	$\{U \rightarrow \{USR, DESK, HALL, SRV\}\}$
input(!secret) @VAULT	$\{key \rightarrow \{KEY\}, secret \rightarrow \{Secret\}\}$	$\{U \rightarrow \{KEY, Secret\}\}$	$\{U \rightarrow \{USR, DESK, HALL, SRV, VAULT\}\}$
move(HALL)	$\{key \rightarrow \{KEY\}, secret \rightarrow \{Secret\}\}$	$\{U \rightarrow \{KEY, Secret\}\}$	$\{U \rightarrow \{USR, DESK, HALL, SRV, VAULT\}\}$
move(USR)	$\{key \rightarrow \{KEY\}, secret \rightarrow \{Secret\}\}$	$\{U \rightarrow \{KEY, Secret\}\}$	$\{U \rightarrow \{USR, DESK, HALL, SRV, VAULT\}\}$

Figure 3.24: Running analysis1

HALL $::^{\langle * \rightarrow \mathsf{m} \rangle}$ [**nil**] ||

JAN $::^{\langle \mathrm{J} \rightarrow \mathsf{m} \rangle}$ [**nil**] ||

USR $::^{\langle \mathrm{U} \rightarrow \mathsf{m} \rangle}$ [input(!key)@DESK.move(HALL)
.move(SRV).input(!Secret)@VAULT.move(HALL).move(USR).nil]$^{\langle \mathbf{U}, \{\} \rangle}$ ||

DESK $::^{\langle \mathrm{U} \rightarrow \mathsf{i} \rangle}$ $\langle KEY^{\{\}} \rangle$ ||

SRV $::^{\langle * \rightarrow \mathsf{m} \rangle}$ [**nil**] ||

VAULT $::^{\langle * \rightarrow \mathsf{i,r} \rangle}$ $\langle Secret^{\{\}} \rangle$ ||

Figure 3.25: The Running Example in insCalc

CHAPTER 4

Extensions to The Insider Framework

In Chapter 3 we developed an insider framework for specifying systems and performing insider analysis. In this chapter we extend the framework to allow all actions to be logged. Each location or datum can now specify, which actions on them are logged, and which are not logged. Some actions may therefore go unnoticed. The extension does not effect the first analysis presented in the previous chapter (Analysis0) but only the latter one. We do no longer have perfect information about the actions taken by each actor, and will have to extend the analysis accordingly. The log files will make the system more realistic, and more importantly will make the analysis result more realistic. The idea is that given a log file, after an incident has occurred, the new analysis will be able to reconstruct a sequence of actions performed by the actors in the system, and thus be able to find (or at least narrow down) the possibilities of finding the insider. In this setting Analysis1 can be thought as an analysis where every single action was logged, and the process definitions as a sequences of actions reconstructed from such a log file. With the logged actions, the system designer must explicitly mark each access mode at locations or data as being logged, and the log file can thus contain "holes" where the actions of actors are not registered. The final analysis that we define will work on these log files, reconstruct the sequence of actions performed by an actor, and fill in the gap between log points to simulate every action that might have happened in between log points.

4.1 Abstract System with Logging

We begin by extending the abstract system with a log component. The abstract system should now be extended with a *global system clock* T with the type $\mathbb{N}$, which is a global clock that enables us to give each entry in the log file a unique label. The log file is modeled by a mapping $\mathsf{Log} : \mathbb{N} \times (\mathsf{Actors} \cup \mathsf{Data} \cup \mathsf{Loc}) \times \mathsf{Loc} \times \mathsf{Loc} \times (\mathsf{LocAccMode} \cup \mathsf{DataAccMode}) \longrightarrow \mathcal{P}(\mathsf{Res})$ and it collects log file entries. The choice of the Log component will be described in Section 4.4. We also add the logged actions to the set of restrictions Res. If we assume that R is the set of restrictions in the system, then the set of access modes is now defined as

$$\mathsf{Res} = \bigcup \{r, \bar{r} \mid r \in \mathsf{R}\}$$

where $\bar{r}$ is the logged action for r. Our definition for a system is changed to

Definition 4.1 (Logged 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, $\mathcal{D} : \mathsf{Loc} \rightarrow \mathsf{Dom}$ a mapping from locations to domains, Cap a set of capabilities, Res a set of restrictions, $\mathcal{C} : \mathsf{Actors} \rightarrow \mathcal{P}(\mathsf{Cap})$ a mapping from actors to capabilities, $\mathcal{R} : (\mathsf{Loc} \cup \mathsf{Data}) \rightarrow \mathcal{P}(\mathsf{Res})$ a mapping from actors and locations to restrictions, $\mathcal{G} : \mathsf{Actors} \rightarrow \mathsf{Loc}$ a mapping from actors to locations and for each restriction r, let $\Phi_r : \mathsf{Cap} \rightarrow \{true, false\}$ be a checker. Let T be a global clock with type $\mathbb{N}$ and $\mathsf{Log} : \mathbb{N} \times (\mathsf{Actors} \cup \mathsf{Data} \cup \mathsf{Loc}) \times \mathsf{Loc} \times \mathsf{Loc} \times (\mathsf{LocAccMode} \cup \mathsf{DataAccMode}) \longrightarrow \mathcal{P}(\mathsf{Res})$ be a logging component. Then we call $\mathcal{S} = \langle \mathcal{I}, \mathsf{Actors}, \mathsf{Data}, \mathcal{D}, \mathcal{G}, \mathcal{C}, \mathcal{R}, \Phi, \mathsf{T}, \mathsf{Log} \rangle$ a logged system.

4.2 Access Modes

Having the log-file extension in place for the abstract system, we now look at how to extend access modes in insCalc. To model the logging of actions, we simply add a new logged access mode for each of the unlogged modes, to represent that the action performed will be logged. It might seem strange to add a logged decrypt mode, as decryption of data can be considered as a hard-to-observe action that is performed privately by the actor, but in the case of machines used for decryption of data it should be possible to offer logging of decryption. Having introduced logged actions, we must ensure that a policy does not contain a logged action along with its non-logged counterpart. The new modes are given in Figure 4.1.

$$
\begin{array}{rcl}
\mathsf{LoggedLocMode} & = & \{\bar{\mathsf{i}}, \bar{\mathsf{r}}, \bar{\mathsf{o}}, \bar{\mathsf{e}}, \bar{\mathsf{m}}\} \\
\mathsf{UnloggedLocMode} & = & \{\mathsf{i}, \mathsf{r}, \mathsf{o}, \mathsf{e}, \mathsf{m}\} \\
\pi_\ell \subseteq \mathsf{LocAccMode} & = & \mathsf{LoggedLocMode} \cup \mathsf{UnloggedLocMode} \\
\mathsf{LoggedDataMode} & = & \{\bar{\mathsf{d}}\} \\
\mathsf{UnLoggedDataMode} & = & \{\mathsf{d}\} \\
\pi_\delta \subseteq \mathsf{DataAccMode} & = & \mathsf{LoggedDataMode} \cup \mathsf{UnLoggedDataMode} \\
\kappa \subseteq \mathsf{Data} & = & \{\text{data used as keys}\} \\
\delta \in \mathsf{LocPolicy} & = & (\mathsf{Loc} \cup \mathsf{Actors} \cup \mathsf{Data} \cup \{*\}) \rightarrow R, R \in \mathcal{P}\{\mathsf{LocAccMode}\} \\
 & & \text{such that } \forall r \in R \Rightarrow \bar{r} \notin R \vee \forall \bar{r} \in R \Rightarrow r \notin R \\
\rho \in \mathsf{DataPolicy} & = & (\mathsf{Loc} \cup \mathsf{Actors} \cup \mathsf{Data} \cup \{*\}) \rightarrow R, R \in \mathcal{P}\{\mathsf{DataAccMode}\} \\
 & & \text{such that } \forall r \in R \Rightarrow \bar{r} \notin R \vee \forall \bar{r} \in R \Rightarrow r \notin R
\end{array}
$$

Figure 4.1: Access Control with Logged Access Modes

4.3 Syntax of The System Specification Language

The update of the System Specification Language reflects the change in the access modes and is straight forward, we simply add the new modes to the syntax. The new modes have an underscore attached to them i.e., **mode_**. The underscore character is used to represent the bar above the mode. The changes to the syntax are presented in Figure 4.2.

LocAccMode	::=	`i`	destructive read
	|	`i_`	destructive read, logged
	|	`r`	non-destructive read
	|	`r_`	non-destructive read, logged
	|	`o`	output a datum
	|	`o_`	output a datum, logged
	|	`e`	spawn a process
	|	`e_`	spawn a process, logged
	|	`m`	move
	|	`m_`	move, logged
DataAccMode	::=	ϵ	empty access
	|	`d`	decrypt
	|	`d_`	decrypt, logged

Figure 4.2: System Specification Language Update

4.4 Semantics

The semantics for the log-file extension is similar to the semantics described for insCalc in Section 3.9. The main difference is the additional global timer T and the logging component Log. The big question when designing the semantics is how to model a realistic log file. We would want to log as much as we possibly can, such as *actor name*, *action*, *from location*, *to location*, and of course *time-stamp*. There should be no problem in logging all this information, as all the information is available in each semantic rule. The question is how realistic it is to register the actor in all cases, as an actor could have been granted access based on something other than his identity. If an actor uses a PIN-code to move to a location, in reality a cipher lock would probably only register the time and possibly the key used for the access, as the lock would have no way of knowing which actor was granted access. We will continue the extension by logging only by which means the access was granted to keep the framework as realistic as possible. We start by adding two helper functions, *grantby* and *decryptby*, listed in Algorithm 12 and Algorithm 13, that provide the semantics with information on how access was granted. The result can be any of four possible cases:

- Names represents that access was granted based on the identity of the actor or that access was not restricted at all (the access policy was defined with the * element),
- Loc represents that access was granted based on the current location of the actor,
- Data represents that access was granted based on some knowledge of the actor,
- ϵ represent that access was **not** granted.

Notice that in the cases where either a policy is defined with the star element, or a resource is unrestricted, the functions *grantby* and *decryptby* return the identity of the actor that is granted the access.

We can now re-factor our reference monitor semantics by restating *grant* and *decrypt* in terms of the two new functions. The changes in the reference monitor semantics are listed in Figure 4.3.

The semantics for the reduction relation is similar to the semantics in Figure 3.15 and Figure 3.16. The main addition is a global timer T and the global logging component Log. We also have to add additional rules for the logged access modes. The additional rules are listed in Figure 4.4 and Figure 4.4 and all the

Algorithm 12 grantby($\mathsf{Names} \times \mathsf{Loc} \times \mathcal{P}(\mathsf{Data}) \times \mathsf{LocAccMode} \times \mathsf{Loc} \times \mathsf{LocPolicy} \to (\mathsf{Names} \times \mathsf{Data} \times \mathsf{Loc} \times \{\epsilon\})$

```
grantby$(n, l, \kappa, a, l', \delta')$ =
if $\delta' = \emptyset$ then
    return $n$ {There is no restriction on access}
else if $* \in \delta'^{-1}(a)$ then
    return $n$ {The action is unrestricted for all actors (star property)}
else if $a \in \delta'(n)$ then
    return $n$
else if $a \in \delta'(l)$ then
    return $l$
else if $\exists k \in \kappa : a \in \delta'(k)$ then
    return $k$
else
    return $\epsilon$
end if
```

Algorithm 13 decryptby($\mathsf{Names} \times \mathsf{Loc} \times \mathcal{P}(\mathsf{Data}) \times \mathsf{DataAccMode} \times \mathsf{Data} \to (\mathsf{Names} \times \mathsf{Data} \times \mathsf{Loc} \times \{\epsilon\})$

```
decryptby$(n, l, \kappa, a, et^{\rho})$ =
if $\rho = \emptyset$ then
    return $n$ {The data was completely public}
else if $* \in \rho^{-1}(a)$ then
    return $n$ {The data is public for all actors (star property)}
else if $a \in \rho(n)$ then
    return $n$
else if $a \in \rho(l)$ then
    return $l$
else if $\exists l' \in \{l'' | (l, l'') \in \mathsf{Con} \wedge a \in \rho(l'')\}$ then
    return $l$
else if $\exists k \in \kappa : \mathsf{a} \in \rho(k)$ then
    return $k$
else
    return $\epsilon$
end if
```

rules use the *grantby* or *decryptby* functions to discover by which means the access was granted. Apart from the logging of the actions in the semantic rules, they are exactly the same as their non-logging counterparts.

(1)

$$\text{grant} \;:\; \mathsf{Names} \times \mathsf{Loc} \times \mathcal{P}(\mathsf{Data}) \times \mathsf{LocAccMode} \times \mathsf{Loc} \times \mathsf{LocPolicy} \to \{\texttt{true}, \texttt{false}\}$$

$$\text{grant}(n, l, \kappa, a, l', \delta') = \begin{cases} \texttt{true} & \text{if grantby}(n, l, \kappa, a, l', \delta') \neq \epsilon \\ \texttt{false} & \text{otherwise} \end{cases}$$

(2)

$$\text{decrypt} \;:\; \mathsf{Names} \times \mathsf{Loc} \times \mathcal{P}(\mathsf{Data}) \times \mathsf{DataAccMode} \times \mathsf{Data} \to \mathcal{P}(\mathsf{Data})$$

$$\text{decrypt}(n, l, \kappa, a, et^{\rho}) = \begin{cases} \{et^{\emptyset}\} & \text{if decryptby}(n, l, \kappa, a, et^{\rho}) \neq \epsilon \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 4.3: Reference Monitor for Access Control

(OUT_LOG)

$$\frac{\begin{array}{c} l' \in \mathsf{Loc} \quad \mathcal{R}(l') = \delta' \quad [\![t]\!] = et^{\rho} \quad \mathcal{K}' = \mathcal{K}[l' \to \mathcal{K}[l'] \cup et^{\rho}] \\ \mathsf{T} < \mathsf{T}' \quad \boxed{\langle \mathcal{S}, n, \kappa \rangle \rightsquigarrow \langle l, l', \bar{\mathsf{o}} \rangle} \quad \mathsf{Log}' = \mathsf{Log}[\mathsf{T} \mapsto (\text{grantby}(n, l, \kappa, \bar{\mathsf{o}}, l'), l, l', \mathsf{o})] \end{array}}{\mathsf{Log}, \mathsf{T}, \mathcal{S} \vdash l ::^{\delta} [\mathbf{out}(t)@l'.P]^{\langle n, \kappa \rangle} \rightarrowtail \mathsf{Log}', \mathsf{T}', \mathcal{S}' \vdash l ::^{\delta} [P]^{\langle n, \kappa \rangle} \mid\mid l' ::^{\delta'} \langle et^{\rho} \rangle}$$

(IN_LOG)

$$\frac{\begin{array}{c} match([\![T]\!], et^{\rho}) = \sigma \quad \mathcal{K}' = \mathcal{K}[l' \to \mathcal{K}[l'] \setminus et^{\rho}, n \to \mathcal{K}[n] \cup et^{\rho}] \\ \mathsf{T} < \mathsf{T}' \quad \boxed{\langle \mathcal{S}, n, \kappa \rangle \rightsquigarrow \langle l, l', \bar{\mathsf{i}} \rangle} \quad \mathsf{Log}' = \mathsf{Log}[\mathsf{T} \mapsto (\text{grantby}(n, l, \kappa, \bar{\mathsf{i}}, l'), l, l', \mathsf{i})] \end{array}}{\begin{array}{c} \mathsf{Log}, \mathsf{T}, \mathcal{S} \vdash l ::^{\delta} [\mathbf{in}(T)@l'.P]^{\langle n, \kappa \rangle} \mid\mid l' ::^{\delta'} \langle et^{\rho} \rangle \rightarrowtail \\ \mathsf{Log}', \mathsf{T}', \mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n, \kappa \cup \{et^{\rho}\} \rangle} \mid\mid l' ::^{\delta'} [\mathbf{nil}] \end{array}}$$

(READ_LOG)

$$\frac{\begin{array}{c} match([\![T]\!], et^{\rho}) = \sigma \quad \mathcal{K}' = \mathcal{K}[n \to \mathcal{K}[n] \cup et^{\rho}] \\ \mathsf{T} < \mathsf{T}' \quad \boxed{\langle \mathcal{S}, n, \kappa \rangle \rightsquigarrow \langle l, l', \bar{\mathsf{r}} \rangle} \quad \mathsf{Log}' = \mathsf{Log}[\mathsf{T} \mapsto (\text{grantby}(n, l, \kappa, \bar{\mathsf{r}}, l'), l, l', \mathsf{r})] \end{array}}{\begin{array}{c} \mathsf{Log}, \mathsf{T}, \mathcal{S} \vdash l ::^{\delta} [\mathbf{read}(T)@l'.P]^{\langle n, \kappa \rangle} \mid\mid l' ::^{\delta'} \langle et^{\rho} \rangle \rightarrowtail \\ \mathsf{Log}', \mathsf{T}', \mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n, \kappa \cup \{et^{\rho}\} \rangle} \mid\mid l' ::^{\delta'} \langle et^{\rho} \rangle \end{array}}$$

Figure 4.4: Extensions to the Operational Semantics for insCalc, Part I

(DECRYPT_LOG)
$$\frac{\begin{array}{c}\text{decrypt}(n,l,\kappa,[\![t]\!]) = \{et^{\emptyset}\} \quad match([\![T]\!],et^{\emptyset}) = \sigma \quad \mathcal{R}' = \mathcal{R}[et \to \emptyset] \\ \mathsf{T} < \mathsf{T}' \quad \mathsf{Log}' = \mathsf{Log}[\mathsf{T} \mapsto (\text{decryptby}(n,l,\kappa,\bar{\mathsf{d}},[\![t]\!]),l,l,\mathsf{d})]\end{array}}{\mathsf{Log},\mathsf{T},\mathcal{S} \vdash l ::^{\delta} [\mathbf{decrypt}(t,T).P]^{\langle n,\kappa\rangle} \rightarrowtail \mathsf{Log}',\mathsf{T}',\mathcal{S}' \vdash l ::^{\delta} [P\sigma]^{\langle n,\kappa\cup et^{\emptyset}\rangle}}$$

(MOVE_LOG)
$$\frac{\begin{array}{c}l' \in \mathsf{Loc} \quad \mathcal{R}(l') = \delta' \quad \mathsf{Dom}(l) = \mathsf{Dom}(l') \quad \mathcal{G}' = \mathcal{G}[n \to l'] \quad \mathsf{T} < \mathsf{T}' \\ \boxed{\langle \mathcal{S},n,\kappa\rangle \rightsquigarrow \langle l,l',\bar{\mathsf{m}}\rangle} \quad \mathsf{Log}' = \mathsf{Log}[\mathsf{T} \mapsto (\text{grantby}(n,l,\kappa,\bar{\mathsf{m}},l'),l,l',\mathsf{m})]\end{array}}{\mathsf{Log},\mathsf{T},\mathcal{S} \vdash l ::^{\delta} [\mathbf{move}(l').P]^{\langle n,\kappa\rangle} \rightarrowtail \mathsf{Log}',\mathsf{T}',\mathcal{S}' \vdash l ::^{\delta} [\mathbf{nil}] \mid\mid l' ::^{\delta'} [P]^{\langle n,\kappa\rangle}}$$

(EVAL_LOG)
$$\frac{\begin{array}{c}l' \in \mathsf{Loc} \quad \mathcal{R}(l') = \delta' \quad \mathsf{Dom}(l') = digital \quad n' \notin \mathsf{Actors} \\ \mathsf{T} < \mathsf{T}' \quad \boxed{\langle \mathcal{I},n,\kappa\rangle \rightsquigarrow \langle l,l',\bar{\mathsf{e}}\rangle} \quad \mathsf{Log}' = \mathsf{Log}[\mathsf{T} \mapsto (\text{grantby}(n,l,\kappa,\bar{\mathsf{e}},l'),l,l',\mathsf{e})]\end{array}}{\begin{array}{c}\mathsf{Log},\mathsf{T},\mathcal{S} \vdash l ::^{\delta} [\mathbf{eval}(n',Q)@l'.P]^{\langle n,\kappa\rangle} \rightarrowtail \\ \mathsf{Log}',\mathsf{T}',\mathcal{S}' \vdash l ::^{\delta} [P]^{\langle n,\kappa\rangle} \mid\mid l' ::^{\delta'} [Q]^{\langle n',\kappa\rangle}\end{array}}$$

Figure 4.5: Extensions to the Operational Semantics for insCalc, Part II

4.5 Analysis with Logging (Analysis2)

In this section we describe the final analysis for the extended version of our insider framework. In contrast to the previous analyses we have dropped the unrealistic assumption that all actions in the system are logged, and will base our analysis on a log file with incomplete information about actions performed in the system. The analysis will have to take into account the actions that are logged, and simulate all possible actions between log points for each and every actor in the system. The system specification will not need to be mapped to a insCalc program, as the analysis will run on the log file as input and not an insCalc program. The analysis is a graph-based algorithm and not a syntax directed as the one in Analysis1. However, the analysis still has to be guided by the semantics of insCalc to be able to make the right decisions and to simulate logged actions. The result of the analysis is the set of data each actor can obtain in the system given a log file of events.

4.5.1 Example System

We will start by presenting an example system with logged actions. Figure 4.6 shows a system where the janitor is initially located at the janitor's workshop, and a user is initially located at the user office. The janitor holds the key to the vault, where some secret information is stored. The system only logs the movement into the janitor's workshop, into the user office, and into the server room and, then the retrieval of the data located in the vault.

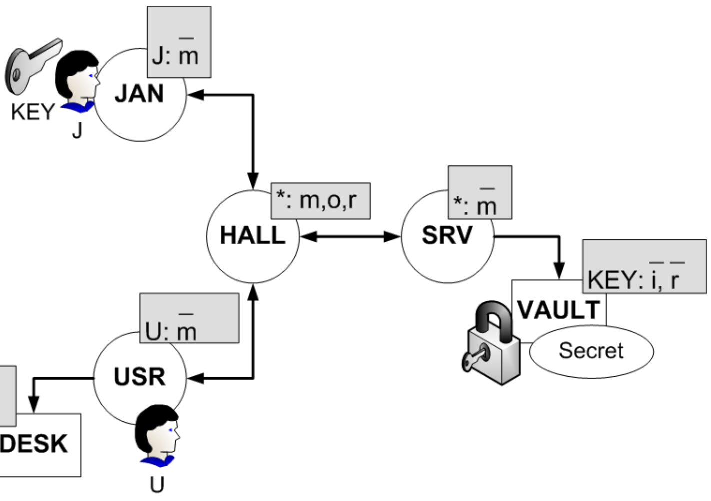


Figure 4.6: The running example

We assume that the following sequence of actions has occurred in the system: The janitor enters his workshop, the user enters the server room, the secret data item is taken from the vault, the user enters his office again, the secret data item is placed on the desk in the user office. The log file is presented in Figure 4.7. The log file shows only that the user was able to go to the server room, open the vault, and retrieve the secret data, return to his office, and place the secret item on his desk. The janitor returned to his workshop before the user entered the server room to get the secret data. We assume that $t_k < t_l$ if $k < l$. If movement to the server room was not logged the janitor could have retrieved the secret data and dropped it in the hall for the user to pick up or even just read the secret data before giving the key to the user. But because the user is the only actor that enters the server room he is the only one that could have taken the secret data. The important part in this example is the *non-logged* action, where the janitor gives the user the key to the vault, and that is exactly the kind of information we want our analysis to find out.

4.5.2 Algorithm for Analysis2

We now develop an algorithm for computing an over-approximation of the data an actor can obtain in a system given a log file of events. In the algorithm we need to keep track of three sets:

- $data : \{\mathsf{Loc} \rightarrow \mathcal{P}(\mathsf{Data})\}$ the set of data at each location,
- $keys : \{\mathsf{Actors} \times \mathsf{Loc} \rightarrow \mathcal{P}(\mathsf{Data})\}$ the key set of each actor at possible locations,
- $locs : \{\mathsf{Actors} \rightarrow \mathcal{P}(\mathsf{Loc})\}$ the set of location that each actor can be at, at a given time.

Initially the set*data* is $\mathcal{K}(\mathsf{Loc})$, i.e., the data stored at each location. The set *locs* is initialized with the current location of each actor, and the set *keys* is defined as κ for each actor. The algorithm processes the log file in an ascending order according to the time stamps. The algorithm simulates all actions that are not logged, but accessible for every user. We call the locations where such actions can take place *log-equivalent locations*. The analysis is listed in Algorithm 14 through Algorithm 17.

The algorithm starts with the function *analysis2*, which holds the main loop for the algorithm. It initializes the three sets of information it keeps track of, before it starts processing the log file one entry at a time. Before each entry in the log file is processed, the function calls *logEquivalent()*, which simulates all unlogged actions in the log-equivalent locations that exist.

The *processLogEntry* function processes a single entry in the log file. It discovers by which means the access was granted, i.e., either by identity, key, or location, and proceeds accordingly. If the function can identify a single user as the one responsible for the log entry, his data structures are updated accordingly. If it is not possible to identify a single actor causing the log file entry, the algorithm simulates the action for all potential actors that could have caused the entry.

$$
\begin{array}{l}
(t_1, J, HALL, JAN, m)\\
(t_2, U, HALL, SRV, m)\\
(t_3, KEY, SRV, VAULT, i)\\
(t_4, U, HALL, USR, m)\\
(t_5, U, USR, DESK, o)
\end{array}
$$

Figure 4.7: Log file

The *logEquivalent* function simulates all unlogged actions for all actors repeatedly, until the values in the three sets do not change anymore, i.e., the sets reach a fixed point. The reason for running the simulation repeatedly is to simulate all possible interleavings of actors and actions. It is in fact a common pattern in program analysis to repeat different paths of execution in order to find the over-approximation of running all possible interleavings of the paths.

The *simulateAction* function is responsible for simulating an action, i.e., to update the data structures according to the simulated action. If the action is a **move**, the location that the actor moves to is added to his *locs* set, and his key set at the given location is updated as well. As soon as he is moved to a new location, the function tries to decrypt all the keys in his key set as there could be encrypted keys that the user could decrypt in the given location. If the action is an **in** or **read** action we have to assume that the actor reads all data in the location, and thus we add all the data at the location to his key set. The algorithm does never *remove* anything from the location, as it tries to be as pessimistic as possible. As soon as the actor has consumed the new data, the algorithm tries to decrypt every single data in his key set, as there could be encrypted data that could be decrypted by the new data.

Algorithm 14 Analysis2

```
analysis2($logFile, \mathcal{S}$) =
for all $n \in$ Actors do
    locs(n) := $\mathcal{G}(n)$
    for all $l \in$ Loc do
        keys(n,l) := $\mathcal{K}(l)$ {Keys is initialized to the actors key set}
    end for
end for
for all $l \in$ Loc do
    data(l) := $\mathcal{K}(l)$
end for
while logFile not empty do
    logEquivalent()
    processLogEntry(getNextEntry(logFile), $\mathcal{S}$)
end while
```

Algorithm 15 processLogEntry

```
processLogEntry$((x, l, l', a), \mathcal{S})$ =
if $x \in$ Actors then
    potentalActors := $\{x\}$
else if $x \in$ Data then
    potentalActors := $\{n \mid x \in keys(n, l) \wedge n \in$ Actors$\}$
else if $x \in$ Loc then
    potentalActors := $\{n \mid x \in locs(n) \wedge n \in$ Actors$\}$
end if
if potentialActors = $\{n\}$ then
    $locs(n) := \{l\}$ {Move the actor to l}
    for all $l'' \in locs(n) : l'' \neq l$ do
        $keys(n, l'') := \mathcal{K}(n)$ {initialize his keys on other location}
    end for
end if
for all $n \in potentialActors$ do
    simulateAction$(n, l, l', a, keys(n, l), \mathcal{S})$
end for
```

Algorithm 16 logEquivalent

```
logEquivalent$(\mathcal{S})$ =
repeat
    $changed := false$
    for all $n \in$ Actors do
        for all $l \in locs(n)$ do
            for all $l' \in$ Loc $\wedge$ $(l, l') \in$ Con do
                for all $a \in unlogged$ : grant$(n, l, a, keys(n, l), l')$ do
                    $changed := changed \vee$ simulateAction$(n, l, a, keys, l')$
                end for
            end for
        end for
    end for
until $changed = false$
```

4.5.3 Running the algorithm

We can now map the system in Figure 4.6 to the abstract system and run the algorithm on the log file listed in Figure 4.7. The table has three columns:

1. **Actor:** showing which actor the rest of the line applies to,

Algorithm 17 simulateAction

```
1: simulateAction$(n, l, a, \kappa, l') =$
2: if $a \in \{\mathsf{m}, \mathsf{e}\}$ then
3:     if $l' \notin locs(n)$ then
4:         $locs(n) := locs(n) \cup \{l'\}$
5:         $keys(n, l') := keys(n, l) \cup$ decryptAll$(n, l', keys(n, l))$
6:         return $true$
7:     end if
8: else if $a = \mathsf{o}$ then
9:     $changes := keys(n, l)$
10:    if $changes \not\subseteq data(l')$ then
11:        $data(l') := data(l') \cup changes$
12:        return $true$
13:    end if
14: else if $a \in \{\mathsf{i}, \mathsf{r}\}$ then
15:    $changes := data(l)$
16:    if $changes \not\subseteq keys(n, l)$ then
17:        $keys(n, l) := keys(n, l) \cup changes$
           $\cup$ decryptAll$(n, l, l', keys(n, l) \cup changes)$
18:        return $true$
19:    end if
20: end if
21: return $false$
```

2. **locs:** showing the contents of the set *locs*, which contains the possible location of each actor,

3. **keys:** showing the contents of the set *keys*, which contains the keys of each actor,

The result of the run is listed in Figure 4.8. Step one is the initialization of the data structures in the algorithm. Step two is after the first call to *logEquivalent*, at this stage the actors move to the hall and the user learns the key from the janitor. Step three is after processing the first log entry. Step four is after the second call to *logEquivalent* and just before the processing of the second entry in the log file. Step five is where the user moves to the server room. Step six is where the user takes something from the vault, and the algorithm must assume that he takes everything. Step seven is where the user returns to the hall. Step eight we simulate everything that could happen and here we must assume that he outputs the secret and thus the janitor also learns the secret. The final step is when the user places the secret data element to his desk. The analysis shows clearly who took the data from the vault, but it cannot tell whether the janitor learned the secret or not, it can only assume the worst, namely that the user gave away the secret in the hall. In order to make the framework even more realistic, one could add probabilities to the actions of actors, e.g., the probability of the user giving away the secret to the janitor could be 0.1 or simply 0.0. That would produce a smaller result set when analyzing the log file, and probably a more realistic one. We see this as future work.

#	*actor*	*locs*	*keys*
1	U	USR	$\{\}$
	J	JAN	$\{JAN \rightarrow \{KEY\}\}$
2	U	$USR, HALL$	$\{HALL \rightarrow \{KEY\},$ $USR \rightarrow \{KEY\}$
	J	$JAN, HALL$	$\{JAN \rightarrow \{KEY\},$ $HALL \rightarrow \{KEY\}$
3	U	$USR, HALL$	$\{HALL \rightarrow \{KEY\},$ $USR \rightarrow \{KEY\}$
	J	JAN	$\{JAN \rightarrow \{KEY\}\}$
4	U	$USR, HALL$	$\{HALL \rightarrow \{KEY\},$ $USR \rightarrow \{KEY\}$
	J	$JAN, HALL$	$\{JAN \rightarrow \{KEY\},$ $HALL \rightarrow \{KEY\}\}$
5	U	SRV	$\{SRV \rightarrow \{KEY\}\}$
	J	$JAN, HALL$	$\{JAN \rightarrow \{KEY\},$ $HALL \rightarrow \{KEY\}\}$
6	U	SRV	$\{SRV \rightarrow \{KEY, Secret\}\}$
	J	$JAN, HALL$	$\{JAN \rightarrow \{KEY\},$ $HALL \rightarrow \{KEY\}\}$
7	U	$HALL$	$\{HALL \rightarrow \{KEY, Secret\}\}$
	J	$JAN, HALL$	$\{JAN \rightarrow \{KEY\},$ $HALL \rightarrow \{KEY\}\}$
8	U	$HALL, USR$	$\{HALL \rightarrow \{KEY, Secret\},$ $USR \rightarrow \{KEY, Secret\}\}$
	J	$JAN, HALL$	$\{JAN \rightarrow \{KEY, Secret\},$ $HALL \rightarrow \{KEY, Secret\}\}$
9	U	USR	$\{USR \rightarrow \{KEY, Secret\}\}$
	J	$JAN, HALL$	$\{JAN \rightarrow \{KEY, Secret\},$ $HALL \rightarrow \{KEY, Secret\}\}$

Figure 4.8: Running analysis2

Chapter 5

Program Design and Implementation

This chapter discusses the design and implementation of a tool for insider analysis. The tool was implemented in the F# and C# programming languages using Visual Studio 2005 on the Windows operating system.

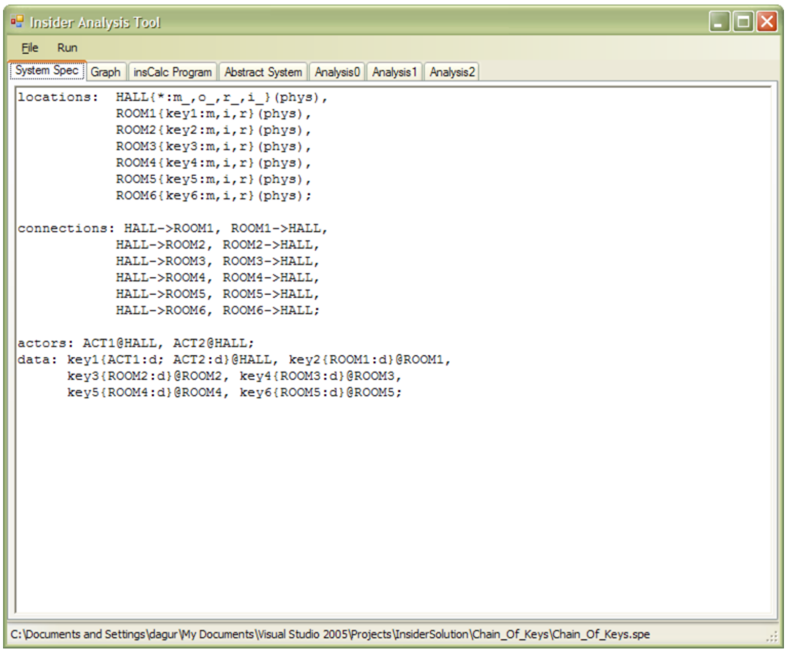
5.1 User Interface

The user interface is a standard Windows Forms application developed in Visual Studio. A good coverage of Windows Forms programming, in C#, is given by Eric Brown in [6]. The tool has the following seven tab pages, that each display a representation of a system or results of analyses:

- **System Spec** - shows the loaded system in the system specification language syntax,
- **Graph** - shows the spatial structure of the loaded system as a image,
- **Abstract System** - shows an internal representation of the system as an abstract system,

- **insCalc Program** - shows the mapping from the system specification to an insCalc program where process variables are placed at each location which contains an actor,
- **Analysis0** - the result of running analysis0 for all actors in the system, from their initial location.
- **Analysis1** - the result of running analysis1 after substituting process variables out for process definitions in the system. The tab page shows both the result window and the process definitions,
- **Analysis2** - the result of running analysis2 on a given log file. This tab page shows both the log file and the result of the analysis.

The tool also has a standard menu bar where the user can load system specifications into the system and run the analyses. For Analysis1 and Analysis2 the user is prompted for a process definition file and a log file respectively. Figure 5.1 shows the tool after a system has been loaded.



```
locations:  HALL{*:m_,o_,r_,i_}(phys),
            ROOM1{key1:m,i,r}(phys),
            ROOM2{key2:m,i,r}(phys),
            ROOM3{key3:m,i,r}(phys),
            ROOM4{key4:m,i,r}(phys),
            ROOM5{key5:m,i,r}(phys),
            ROOM6{key6:m,i,r}(phys);

connections: HALL->ROOM1, ROOM1->HALL,
            HALL->ROOM2, ROOM2->HALL,
            HALL->ROOM3, ROOM3->HALL,
            HALL->ROOM4, ROOM4->HALL,
            HALL->ROOM5, ROOM5->HALL,
            HALL->ROOM6, ROOM6->HALL;

actors: ACT1@HALL, ACT2@HALL;
data: key1{ACT1:d; ACT2:d}@HALL, key2{ROOM1:d}@ROOM1,
      key3{ROOM2:d}@ROOM2, key4{ROOM3:d}@ROOM3,
      key5{ROOM4:d}@ROOM4, key6{ROOM5:d}@ROOM5;
```

C:\Documents and Settings\dagur\My Documents\Visual Studio 2005\Projects\InsiderSolution\Chain_Of_Keys\Chain_Of_Keys.spe

Figure 5.1: The Running Example as a Specification

5.1.1 Loading Systems

The first thing the user of the tool must do is load a system specification. The system specification has the same syntax as the system specification language presented in Section 3.6. The tool parses the system specification and responds with an error message if the specification can not be parsed. If the specification can be parsed with out problems, a graph of the spatial structure is drawn in the *Graph* tab page, as shown in Figure 5.2.

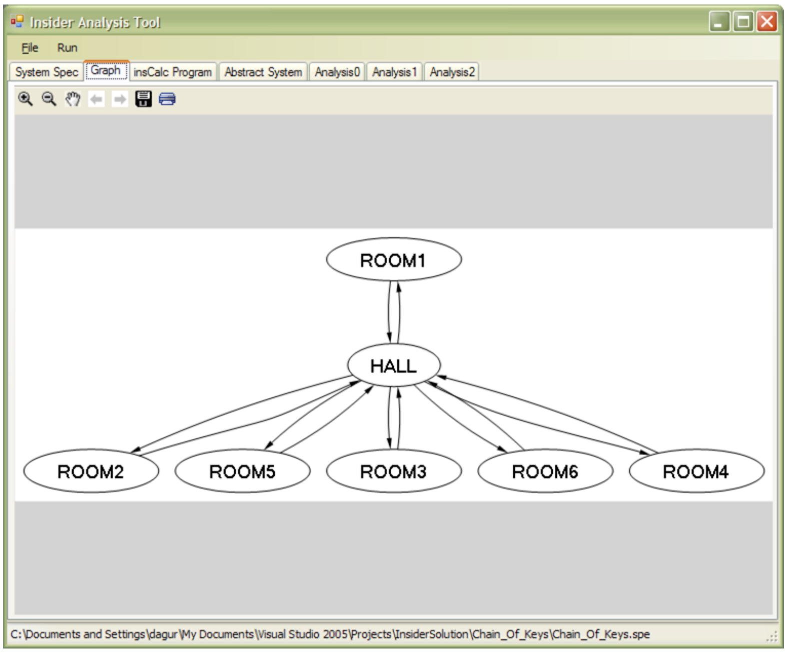


Figure 5.2: Graph Tab Page

The *insCalc Program* tab page displays the insCalc program that is the result of mapping the system to insCalc. The resulting insCalc program contains process variables at locations where actors are present. Immediately after a system is loaded, the tabs in *Analysis0*, *Analysis1*, and *Analysis2* are cleared.

5.1.2 Analysis0

Analysis0 can be executed as soon as a system has been loaded into the tool. The analysis finds reachable locations and data for each actor in the system,

```
Analysis0 Results for Actor ACT1 Located at HALL
-----------------------------------------------------------------
Reachable Locations: { HALL, ROOM1, ROOM2, ROOM3, ROOM4, ROOM5,
                       ROOM6 }
Reachable Data: { key1{}, key1{ACT1: d, ACT2: d}, key2{},
                  key2{ROOM1: d}, key3{}, key3{ROOM2: d}, key4{},
                  key4{ROOM3: d}, key5{}, key5{ROOM4: d}, key6{},
                  key6{ROOM5: d} }


Analysis0 Results for Actor ACT2 Located at HALL
-----------------------------------------------------------------
Reachable Locations: { HALL, ROOM1, ROOM2, ROOM3, ROOM4, ROOM5,
                       ROOM6 }

Reachable Data: { key1{}, key1{ACT1: d, ACT2: d}, key2{},
                  key2{ROOM1: d}, key3{}, key3{ROOM2: d}, key4{},
                  key4{ROOM3: d}, key5{}, key5{ROOM4: d}, key6{},
                  key6{ROOM5: d} }
```

Figure 5.3: Analysis0 Result Report

from their initial locations. The output for the analysis is a text report listing up data, and locations that the actors can discover by roaming the system, as shown in Figure 5.3.

5.1.3 Analysis1

For analysis1 the user must provide the tool with an input file, with process definitions, as defined in Section 3.10.2.2. The result tab page is divided into two sections, the upper part shows the process definitions used in the analysis, and the lower part shows the result of the analysis. The process variables, in the insCalc program, are substituted out for the process definitions supplied in the input file. It is important that all process definitions end with a **nil** action for the parser to be able to parse the file. No error message is produced if the actor is not able to perform an action in the process sequence, i.e., the process definition is not validated before it is run.

After the substitution the analysis is executed, and the result of the analysis is presented in the lower text box. The result is text report showing the contents of $\hat{L}$, $\hat{\kappa}$, $\hat{T}$, and $\hat{\sigma}$, which denote reachable locations, reachable data, data at locations, and data in variables respectively. Figure 5.4 shows the tab page for analysis1.

```
Reachable Locations:
ACT1 = {HALL, ROOM1, ROOM2, ROOM3, ROOM4, ROOM5, ROOM6}
ACT2 = {HALL}

Reachable Data:
ACT1 = {key1{}, key1{ACT1: d, ACT2: d}, key2{}, key2{ROOM1: d},
        key3{}, key3{ROOM2: d}, key4{}, key4{ROOM3: d}, key5{},
        key5{ROOM4: d}, key6{}, key6{ROOM5: d}}
ACT2 = {}

Data at locations:
HALL = {key1{ACT1: d, ACT2: d}}
ROOM1 = {key2{ROOM1: d}}
ROOM2 = {key3{ROOM2: d}}
ROOM3 = {key4{ROOM3: d}}
ROOM4 = {key5{ROOM4: d}}
ROOM5 = {key6{ROOM5: d}}
ROOM6 = {}

Data at variables:
dec_key6 = {key6{}}
key6 = {key6{ROOM5: d}}
dec_key5 = {key5{}}
key5 = {key5{ROOM4: d}}
dec_key4 = {key4{}}
key4 = {key4{ROOM3: d}}
dec_key3 = {key3{}}
key3 = {key3{ROOM2: d}}
dec_key2 = {key2{}}
key2 = {key2{ROOM1: d}}
dec_key1 = {key1{}}
key1 = {key1{ACT1: d, ACT2: d}}
```

Figure 5.4: Analysis1 Result Report

5.1.4 Analysis2

For analysis2 the user must provide the tool with a log file. The log file is loaded into the upper text box in the Analysis2 tab page, and the result of the analysis is displayed in the lower text box. The analysis processes the log file entries in the same order as in the log file, and produces a text report showing reachable locations, data that each actor can have at each location in the system, and finally data located at each location in the system. The result tab page for analysis2 is shown in Figure 5.5.

```
Locations for Actors:
ACT1 = {HALL, ROOM1, ROOM2, ROOM3, ROOM4, ROOM5, ROOM6}
ACT2 = {HALL, ROOM1, ROOM2, ROOM3, ROOM4, ROOM5, ROOM6}

Reachable Data:
ACT1:
HALL = {key1{}, key1{ACT1: d, ACT2: d}, key2{}, key2{ROOM1: d},
        key3{}, key3{ROOM2: d}, key4{}, key4{ROOM3: d}, key5{},
        key5{ROOM4: d}, key6{}, key6{ROOM5: d}}
ROOM1 = {key1{}, key1{ACT1: d, ACT2: d}, key2{}, key2{ROOM1: d},
        key3{}, key3{ROOM2: d}, key4{}, key4{ROOM3: d}, key5{},
        key5{ROOM4: d}, key6{}, key6{ROOM5: d}}
ROOM2 ={key1{}, key1{ACT1: d, ACT2: d}, key2{}, key2{ROOM1: d},
        key3{}, key3{ROOM2: d}, key4{}, key4{ROOM3: d}, key5{},
        key5{ROOM4: d}, key6{}, key6{ROOM5: d}}
...

ACT2:
...

Data at Locations:
HALL = {key1{}, key1{ACT1: d, ACT2: d}, key2{},
        key2{ROOM1: d}, key3{}, key3{ROOM2: d}, key4{},
        key4{ROOM3: d}, key5{}, key5{ROOM4: d}, key6{},
        key6{ROOM5: d}}
ROOM1 = {key2{ROOM1: d}}
ROOM2 = {key3{ROOM2: d}}
ROOM3 = {key4{ROOM3: d}}
ROOM4 = {key5{ROOM4: d}}
ROOM5 = {key6{ROOM5: d}}
ROOM6 = {}
```

Figure 5.5: Analysis2 Result Report

5.2 Program Source

5.2.1 Programming Languages

The program is written in C# and F# using Visual Studio 2005 on the Windows operating system. F# is a functional language in the ML language family, and it is developed by Microsoft Research. The language is strongly typed, and is much similar to the Ocaml [17] language. The reason for choosing F# is that the support for creating parsers and lexical analyzers is excellent in F#, and data structures for mathematical concepts are easy to express in ML languages. Also the pattern matching features of the languages gives a much shorter code and is thus easier to debug and maintain. There is a F# plug-in for Visual Studio, which is not perfect, but in times of trouble there is an excellent community web site where programmers get quick respond to problems they run into [7]. Our experience using F# is generally good and the success of the implementation, of our tool, is much due to the choice of language. The communication between the two languages C# and F# was sometimes strange, and we had to create strange work-a-rounds to make the code work. F# is definitely a promising language and it is nice to finally have a functional language that can be used side-by-side with the industrial strength languages.

5.2.2 Programming Environment

As mentioned the tool was developed in Visual Studio 2005 using the F# plug-in. The code implementing the insider framework was all in F#, and most of the time the theory mapped directly to F# code. Of course we found errors in the algorithms when we implemented them, but these were minor bugs that did not change the overall structure of the algorithms. The Visual Studio solution was divided into two projects; the graphical user interface written in C#, and the insider framework dynamic link library written in F#. The debug feature of Visual Studio was a great help when debugging the program.

5.2.3 Program Structure

The program is organized into two Visual Studio projects, one for the user interface, written in C# and one for parsers, analysis and various internal representation of a system, written in F#. The most important part of the program is the project written in F#, as it is the one implementing the Insider Framework.

In the following we will give an overview of the files and the data structures used to implement the framework. The project is divided into twelve major files, and Figure 5.6 shows a dependency graph of how these files are dependent on each other.

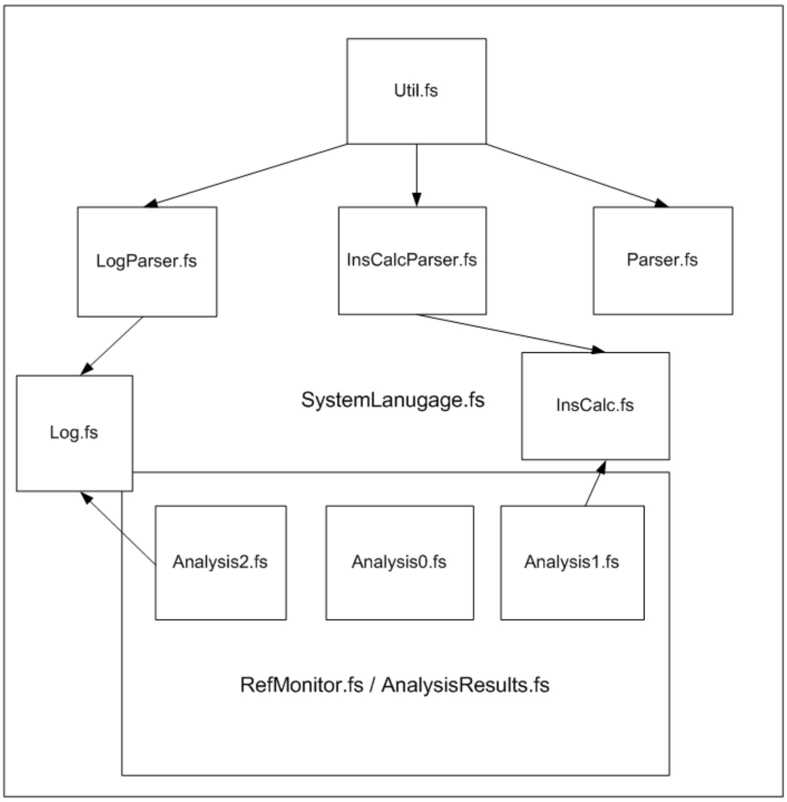


Figure 5.6: Dependency Graph

The **SystemLanguage.fs** file contains the data structures and functions that make up the internal representation of the abstract system. This is the most fundamental file in the project as all other files refer to it. In Figure 5.6 it is shown as covering all other files to demonstrate that all other files are dependent on it. Figure 5.7 shows the data structures for the abstract system, they look like much like the model presented in Section 3.4.

The **Util.fs** file contains three functions, that each parse one of the possible input files. There is parse function for the specification files, a parser for process definition files, and a parser for log files. The three parsers are located in the files **Parser.fs**, **InsCalcParser.fs**, and **LogParser.fs** respectively.

```
type name = string

type location_access = Input | Read | Out | Eval | Move
                     | Input_ | Read_ | Out_ | Eval_ | Move_

type location_policy = LocPolicy of name * location_access list
                     | LocPolicyStar of location_access list

type location = Loc of name * location_policy list

type domain = Dom of name

type connection = Con of name * name

type actor = Act of name

type data_access = Decrypt

type data_policy = DataPolicy of name * data_access
                 | DataPolicyStar of data_access

type data = Data of name * data_policy list
```

Figure 5.7: F# Data Structures for the Abstract System

The **InsCalc.fs** file contains the data structures for the internal representations of insCalc programs. An abstract system can be mapped to an insCalc program, and the process definitions can be plugged into the program using the parser in **InsCalcParser.fs**. The data structures for insCalc programs are listed in Figure 5.8 and the are also strikingly similar to the definitions presented in Figure 3.8.

The **Log.fs** file contains the data structure for representing log files. A log file is represented as a list of log entries.

The **Refmonitor.fs** file is the file implementing the reference monitor semantics, i.e., the *grant*, *decrypt*, *leadsTo*, *encrypt*, and *decryptAll* functions. As the correctness of the analyses depend on the functions in this file, there is also a test file that tests all these functions.

The **AnalysisResults.fs** file contains data structures that hold the results of the analyses. It contains "toString()" functions for the analysis results, to make it possible to print out the results as text.

```
type locality = Local of name | LocVar of name

type formal = Formal of name

type field = Value of name | Variable of name

type template = FormalVar of formal | Field of field

type proc = Nil
          | Action of action * proc
          | Par of proc * proc
          | Inv of name

and action = Out of field * locality
        | In of template * locality
        | Read of template * locality
        | Encrypt of field * data_policy list * formal
        | Decrypt of field * formal
        | Eval of name * proc * name
        | Move of name

type net = ProcNode of location * proc * name * Set<data>
         | NilNode of location
         | TupleNode of location * data
         | CompNode of net * net
```

Figure 5.8: F# Data Structures for the insCalc Syntax

Finally, the files **Analysis0.fs**, **Analysis0.fs**, and **Analysis2.fs** implement Analysis0, Analysis1, and Analysis2 respectively.

5.3 Parsers and Lexers

The parsers and lexical analyzers for the three input files where developed using the tools FSLEX and FSYACC which come with the F# development environment. These tools are very similar to the ocamllex and ocamlyacc tools for the Ocaml language [14], [15]. The syntax in Section 3.7 and Section 3.10.2.2 mapped directly to FSYACC specifications. Detailed information on lexical analyzers and parser generators can be found in [1, 9, 3].

5.4 Summary

This chapter has given the reader an overview of the tool developed for doing insider analysis. We are pleased with the results and believe that our tool runs the three analysis correctly. The choice of language is a big factor in the success of programming the tool, and we believe that F# is a good choice for programming tools, that build on language based research. Visual Studio 2005 is also an excellent programming environment, and its debugging feature is amazing and saves the programmer a lot of time when debugging.

Chapter 6

Evaluation

One of the goals of this project was to develop a tool for performing insider analysis. A description of the implementation of the tool was given in Chapter 5, but in this chapter we will focus on the evaluation of the tool. For the evaluation we use two system specifications, along with process definitions and log files for each system. Each system will demonstrate an important property of the tool, and hopefully demonstrate the correctness of the algorithms in the framework and the implementation of the tool.

6.1 Specification 1

We begin by analyzing a system that we choose to call "Specification 1". It is a simple system with seven rooms connected through a single hall. There are six locations named "Room1" through "Room6" and a location named "Kitchen". In the Kitchen there is a Waste basket containing a document, that can be decrypted by going into the location named "Room4". There are cipher locks on Room5 and Room6, and only an actor holding key2 can enter Room5, and only the one holding the Doc data element can enter Room6. There is a computer in Room1 connected to a printer located in Room2. Figure 6.1 shows a graphical representation of the system and Figure 6.2 shows the system specification loaded into our tool.

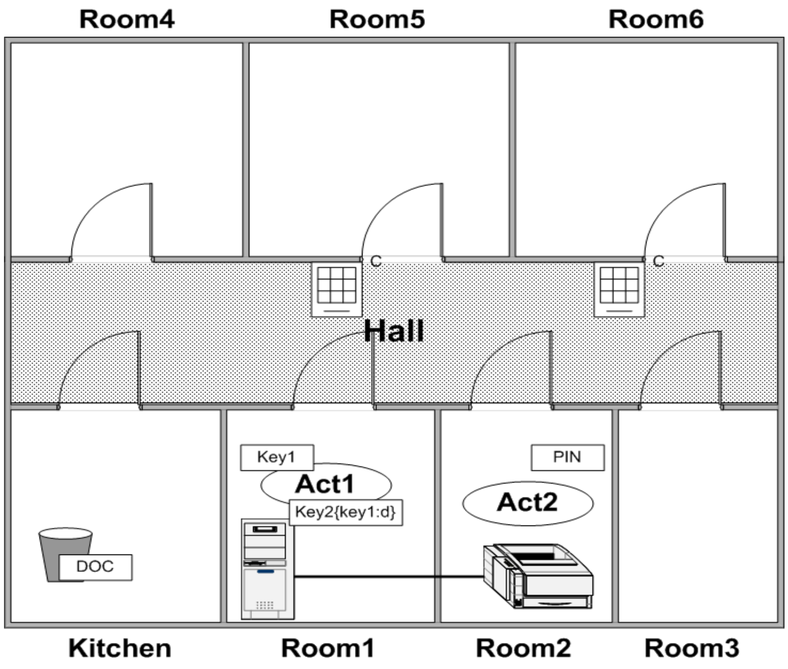


Figure 6.1: Specification 1

6.1.1 Analysis0

We want to demonstrate that Analysis0 makes the actors pick up data and use them as keys, even though the actors have to go to a lot of trouble in decrypting the data.

The analysis should make Act1 decrypt key2 by using key2, and make him go to the Kitchen and pick the Doc from the Waste and decrypt it at Room4. It should also make Act1 explore Room5, which requires key2, and pick up the Pin code located there. Finally, Act1 will explore Room6 which requires the decrypted Doc.

The same thing should happen with Act2, but he will not be able to reach Room5 as he does not have key2 at his disposal, thus he will not be able to retrieve the Pin code located at Room5.

This should hopefully demonstrate that actors can use things they pick up as keys, if they can decrypt them, and that access control works as expected.

```
locations: Hall{*:m_,o,r,i}(phys),
    Room1{*:m}(phys),Room2{*:m}(phys),
    Room3{*:m}(phys),Room4{Hall:m_}(phys),
    Room5{key2:m_}(phys),Room6{Doc:m_}(phys),
    Kitchen{Hall:m_}(phys),
    Waste{*:o,i,r}(phys),
    Pc1{*:e,o,i,r}(dig),
    Printer{*:o,i,r}(dig);
connections:Hall->Room1, Room1->Hall,
        Hall->Room2, Room2->Hall,
        Hall->Room3, Room3->Hall,
        Hall->Room4, Room4->Hall,
        Hall->Room5, Room5->Hall,
        Hall->Room6, Room6->Hall,
        Hall->Kitchen, Kitchen->Hall,
        Kitchen->Waste, Room1->Pc1,
        Room2->Printer, Pc1->Printer,
        Printer->Pc1;
actors: Act1@Room1, Act2@Room2; data: Doc{Room4:d}@Waste,
key1{}@Act1, key2{key1:d}@Act1,
    Pin{}@Room5;
```

Figure 6.2: Specification1: System Specification

Figure 6.3 shows the output of our tool when running Analysis0. Act1 is able to reach all locations in the system and retrieve all data, but Act2 is not able to reach Room5 and retrieve the Pin data item. The result is as we expected, and the running time of the analysis was close to zero seconds, without any delays in the user interface.

6.1.2 Analysis1

For Analysis1 we explicitly state what each actor in the system does. We cannot specify the order things happen so the analysis will simulate all possible interleavings of actions between actors. The analysis is much more precise than analysis0, that only provided us with an upper bound of possible actions. The analysis can be viewed as a reconstruction of actions that where performed in a system, where every single action was observable.

The process definition we will provide will be a simple sequence of action, where Act1 performs all the necessary actions to get access to Room6, i.e., pick up

```
Analysis0 Results for Actor Act1 Located at Room1
---------------------------------------------------------------
Reachable Locations: { Hall, Kitchen, Pc1, Printer, Room1,
                       Room2, Room3, Room4, Room5, Room6,
                       Waste }
Reachable Data: { Doc{}, Doc{Room4: d}, key1{}, key2{},
                  key2{key1: d} }

Analysis0 Results for Actor Act2 Located at Room2
---------------------------------------------------------------
Reachable Locations: { Hall, Kitchen, Pc1, Printer, Room1,
                       Room2,Room3, Room4, Room6, Waste }
Reachable Data: { Doc{}, Doc{Room4: d} }
```

Figure 6.3: Specification1: Analysis0 Results

Doc from the Waste bin, decrypt it at Room4, and then use it to access Room6. We will not provide any action sequence for Act2, in order demonstrate that he will not acquire any knowledge just by standing in Room2 doing nothing. The process definition is shown in Figure 6.4.

```
Act1 := move("Hall").move("Kitchen").
        in(!doc)@"Waste".move("Hall").move("Room4").
        decrypt(doc,!doc_decrypted).move("Hall").
        move("Room6").nil
```

Figure 6.4: Specification1: Process Definitions for Analysis1

The result of running the analysis on process definition is shown in Figure 6.5. The result shows that Act1 can reach the expected locations, and does not explore any other locations which are not given in the process definition. Act2 does not explore any location and does not discover any data. One thing to notice is that even though Act1 does not explicitly decrypt key2, the key is decrypted as soon as he starts moving around. The algorithm decrypt any keys in the actors key set before he starts moving around in the system. When the actor is started moving he must explicitly decrypt all data that he picks up along the way.

```
Reachable Locations:
Act1 = {Hall, Kitchen, Room1, Room4, Room6}
Act2 = {Room2}

Reachable Data:
Act1 = {Doc{}, Doc{Room4: d}, key1{}, key2{},
key2{key1: d}} Act2 = {}

Data at locations:
Hall = {}
Room1 = {}
Room2 = {}
Room3 = {}
Room4 = {}
Room5 = {Pin{}}
Room6 = {}
Kitchen = {}
Waste = {Doc{Room4: d}}
Pc1 = {}
Printer = {}

Data at variables:
doc_decrypted = {Doc{}}
doc = {Doc{Room4: d}}
```

Figure 6.5: Specification1: Analysis1 Results

6.1.3 Analysis2

For Analysis2 we provide the system with a log file of actions, that where observed in the system at locations where actions are logged. The log file is presented in Figure 6.6.

```
(0, Actor(Act1), Room1, Hall, m);
(1, Location(Hall), Hall, Kitchen, m);
(2, Actor(Act1), Kitchen, Hall, m);
(3, Location(Hall), Hall, Room4, m);
(4, Actor(Act1), Room4, Hall, m);
(4, Key(Doc), Hall, Room6, m)
```

Figure 6.6: Specification1: Log file for Analysis2

```
Locations for Actors:
Act1 = {Room6}
Act2 = {Room2}
Reachable Data:
Act1:
Hall = {Doc{}, Doc{Room4: d}, key1{}, key2{}, key2{key1: d}}
Room1 = {} Room2 = {} Room3 = {} Room4 = {} Room5 = {}
Room6 = {Doc{}, Doc{Room4: d}, key1{}, key2{}, key2{key1: d}}
Kitchen = {}
Waste = {}
Pc1 = {}
Printer = {}
Act2:
Hall = {}
Room1 = {}
Room2 = {Doc{}, Doc{Room4: d}, key1{}, key2{}, key2{key1: d}}
Room3 = {} Room4 = {} Room5 = {} Room6 = {}
Kitchen = {}
Waste = {}
Pc1 = {}
Printer = {}
Data at Locations:
Hall = {Doc{}, Doc{Room4: d}, key1{}, key2{}, key2{key1: d}}
Room1 = {} Room2 = {} Room3 = {} Room4 = {}
Room5 = {Pin{}}
Room6 = {}
Kitchen = {}
Waste = {Doc{Room4: d}, key1{}, key2{}, key2{key1: d}}
Pc1 = {Doc{}, Doc{Room4: d}, key1{}, key2{}, key2{key1: d}}
Printer = {Doc{}, Doc{Room4: d}, key1{}, key2{}, key2{key1: d}}
```

Figure 6.7: Specification1: Analysis2 Results

The log file records the same sequence of events as the process definition for Analysis1. The result of running the analysis with the log file is presented in Figure 6.7. The results have three components: reachable locations for each actor, reachable data for each actor at each location, and data located at each location in the system. We can see from the results that Act1 ended his trip at Room6, and that Act2 did not move from Room2. Act1 did manage to decrypt the Doc data element, found in Waste bin in the Kitchen. There are a lot of data in the Waste, PC, and the Printer and this is because the **out** action is not logged at these locations. The analysis must assume that Act1 could have outputted his data at these locations even though he did not.

The analysis seems to do the job correctly, but it is sensitive to errors in the log file. There are few integrity checks on the log file entries and it is easy to make mistake that give unexpected results.

6.2 Specification 2

We call the second system, that we use for evaluation our tool, "the paycheck" system. It is a normal office scenario with four actors, which each have received their paycheck in their PCs. The access annotations for this system are realistic, there is a surveillance camera in the hall that monitors movement in the hall, and the actors can perform all actions at all locations.

We want to demonstrate that especially Analysis2 is too imprecise because it is too pessimistic. It always assumes the worst, that all actors exchange every data they have, if the **out**, **in**, and **read** actions are not logged. Figure 6.8 shows the system graphically and Figure 6.9 shows the system specification file for the system.

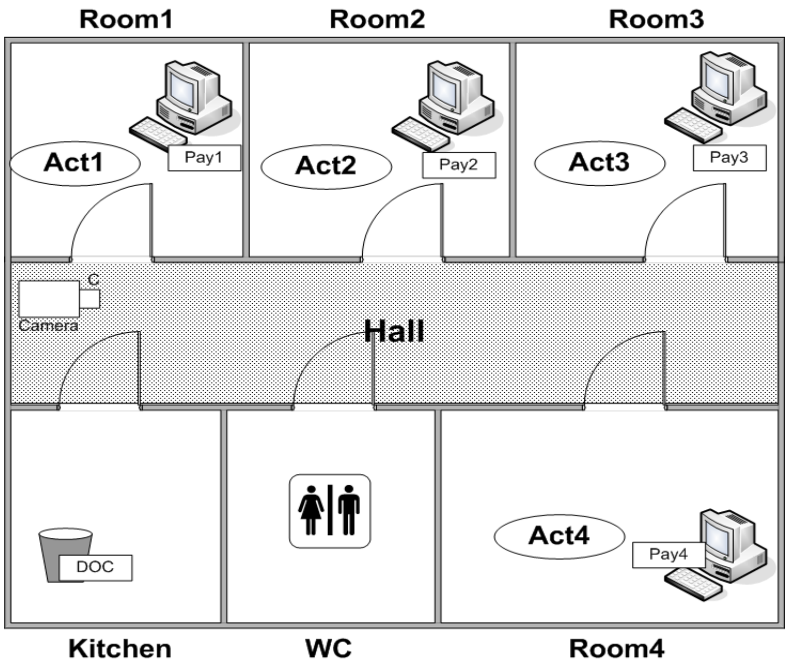


Figure 6.8: Specification 1

```
locations: Hall{*:m_,o,r,i}(phys),
    Room1{*:m,i,o,r}(phys),Room2{*:m,i,o,r}(phys),
    Room3{*:m,i,o,r}(phys),Room4{*:m,i,o,r}(phys),
    Wc{*:m,i,o,r}(phys),
    Kitchen{Hall:m,i,o,r}(phys),
    Waste{Kitchen:o,i,r}(phys),
    Pc1{Act1:e,o,i,r}(dig),
    Pc2{Act2:e,o,i,r}(dig),
    Pc3{Act3:e,o,i,r}(dig),
    Pc4{Act3:e,o,i,r}(dig);
connections:Hall->Room1, Room1->Hall,
        Hall->Room2, Room2->Hall,
        Hall->Room3, Room3->Hall,
        Hall->Room4, Room4->Hall,
        Hall->Wc, Wc->Hall,
        Hall->Kitchen, Kitchen->Hall,
        Kitchen->Waste;
actors: Act1@Room1, Act2@Room2, Act3@Room3, Act4@Room4;
data:Doc{}@Waste;
```

Figure 6.9: Specification2: System Specification

6.2.1 Analysis0

For Analysis0 there are no surprises, each actor can get the Doc data element in the Waste bin in the Kitchen and his paycheck. Each actor can reach all locations. The result of running Analysis0 is shown in Figure 6.10.

```
Analysis0 Results for Actor Act1 Located at Room1
-------------------------------------------------------------
Reachable Locations:
{ Hall, Kitchen, Pc1, Room1, Room2, Room3, Room4, Waste, Wc }
Reachable Data:
{ Doc{}, Pay1{}, Pay1{Act1: d} }

Analysis0 Results for Actor Act2 Located at Room2
-------------------------------------------------------------
Reachable Locations:
{ Hall, Kitchen, Pc2, Room1, Room2, Room3, Room4, Waste, Wc }
Reachable Data:
{ Doc{}, Pay2{}, Pay2{Act2: d} }

Analysis0 Results for Actor Act3 Located at Room3
-------------------------------------------------------------
Reachable Locations:
{ Hall, Kitchen, Pc3, Room1, Room2, Room3, Room4, Waste, Wc }
Reachable Data:
{ Doc{}, Pay3{}, Pay3{Act3: d} }

Analysis0 Results for Actor Act4 Located at Room4
-------------------------------------------------------------
Reachable Locations:
{ Hall, Kitchen, Pc4, Room1, Room2, Room3, Room4, Waste, Wc }
Reachable Data:
{ Doc{}, Pay4{}, Pay4{Act4: d} }
```

Figure 6.10: Specification2: Analysis0 Results

6.2.2 Analysis1

For Analysis1 we provide process definitions, where each actor reads his paycheck and then goes to the Kitchen to meet the other actors. The process definitions are thus the same for all actors, and they are displayed in Figure 6.11.

The result of the analysis is displayed in Figure 6.12, and it contains the expected result. Each actor can reach his office, the Hall, and the Kitchen, and can read his own paycheck.

```
Act1 := in(!pay1)@"Pc1".decrypt(pay1, !pay1_dec).
      move("Hall").move("Kitchen").nil;
Act2 := in(!pay2)@"Pc2".decrypt(pay2, !pay2_dec).
      move("Hall").move("Kitchen").nil;
Act3 := in(!pay3)@"Pc3".decrypt(pay3, !pay3_dec).
        move("Hall").move("Kitchen").nil;
Act4 := in(!pay4)@"Pc4".decrypt(pay4, !pay4_dec).
        move("Hall").move("Kitchen").nil
```

Figure 6.11: Specification2: Analysis1 Process Definitions

```
Reachable Locations:
Act1 = {Hall, Kitchen, Room1}
Act2 = {Hall, Kitchen, Room2}
Act3 = {Hall, Kitchen, Room3}
Act4 = {Hall, Kitchen, Room4}
Reachable Data:
Act1 = {Pay1{}, Pay1{Act1: d}}
Act2 = {Pay2{}, Pay2{Act2: d}}
Act3 = {Pay3{}, Pay3{Act3: d}}
Act4 = {Pay4{}, Pay4{Act4: d}}
Data at locations:
Hall = {} Room1 = {} Room2 = {} Room3 = {} Room4 = {}
Wc = {}
Kitchen = {}
Waste = {Doc{}}
Pc1 = {Pay1{Act1: d}}
Pc2 = {Pay2{Act2: d}}
Pc3 = {Pay3{Act3: d}}
Pc4 = {Pay4{Act4: d}}
Data at variables:
pay4_dec = {Pay4{}}
pay4 = {Pay4{Act4: d}}
pay3_dec = {Pay3{}}
pay3 = {Pay3{Act3: d}}
pay2_dec = {Pay2{}}
pay2 = {Pay2{Act2: d}}
pay1_dec = {Pay1{}}
pay1 = {Pay1{Act1: d}}
```

Figure 6.12: Specification2: Analysis1 Results

6.2.3 Analysis2

For Analysis2 we provide the system with a log file of recorded actions in the system. We want this system to have realistic access-modes on the locations, and have chosen to log only the movement of actors in the Hall. The surveillance camera in the Hall could be connected to a face-recognition program that logs which actors move into the Hall. The camera cannot log what the actors are saying or see what information is being exchanged in the Hall. The log file, we present, shows simply that each actor moves to the Hall and nothing more. The log file is presented in Figure 6.13.

```
(0, Actor(Act1), Room1, Hall, m);
(1, Actor(Act2), Room2, Hall, m);
(2, Actor(Act3), Room3, Hall, m);
(3, Actor(Act4), Room4, Hall, m)
```

Figure 6.13: Specification2: Analysis2 Log File

The output of the analysis is shown in Figure 6.14, and shows that when we run Analysis2 in our tool every actor exchanges every information that is available to him. Every actor also leaves all his data at every location. It is obviously not very realistic, that every actor tells the next what he got payed. The analysis is thus not very realistic when it comes to realistic access annotations. It would be more realistic to add some kind of probability that a given data is exchanged. A simple solution would be to annotate the data with either *private* or *public*. Private data are data that the actor would never give away and public data would be something he might give away.

6.3 Summary

This section contains the evaluation of our tool. It seems to give us the expected results and run the analysis very fast. It is hard to come up with systems to demonstrate the power of the analyses because of the complexity of things that can happen in the system. It is easy to loose track of things in a system with more than a handful of locations, data and actors. The last analysis seem to be a bit unrealistic when not every action is logged.

```
Locations for Actors:
Act1 = {Hall, Kitchen, Pc1, Room1, Room2,
        Room3, Room4, Wc}
Act2 = {Hall, Kitchen, Pc2, Room1, Room2,
        Room3, Room4, Wc}
Act3 = {Hall, Kitchen, Pc3, Room1, Room2,
        Room3, Room4, Wc}
Act4 = {Hall, Kitchen, Pc4, Room1, Room2,
        Room3, Room4, Wc}

Reachable Data:
Act1-Act3:
Hall = {Doc{}, Pay1{}, Pay1{Act1: d}, Pay2{},
       Pay2{Act2: d}, Pay3{}, Pay3{Act3: d},
       Pay4{}, Pay4{Act4: d}}
Room1 = {Doc{}, Pay1{}, Pay1{Act1: d}, Pay2{},
       Pay2{Act2: d}, Pay3{}, Pay3{Act3: d},
       Pay4{}, Pay4{Act4: d}}
Room2 = {Doc{}, Pay1{}, Pay1{Act1: d}, Pay2{},
       Pay2{Act2: d}, Pay3{}, Pay3{Act3: d},
       Pay4{}, Pay4{Act4: d}}
Room3 = {Doc{}, Pay1{}, Pay1{Act1: d}, Pay2{},
       Pay2{Act2: d}, Pay3{}, Pay3{Act3: d},
       Pay4{}, Pay4{Act4: d}}
Room4 = {Doc{}, Pay1{}, Pay1{Act1: d}, Pay2{},
       Pay2{Act2: d}, Pay3{}, Pay3{Act3: d},
       Pay4{}, Pay4{Act4: d}}
Wc = {Doc{}, Pay1{}, Pay1{Act1: d}, Pay2{},
       Pay2{Act2: d}, Pay3{}, Pay3{Act3: d},
       Pay4{}, Pay4{Act4: d}}
Kitchen = {Doc{}, Pay1{}, Pay1{Act1: d}, Pay2{},
       Pay2{Act2: d}, Pay3{}, Pay3{Act3: d},
       Pay4{}, Pay4{Act4: d}}
Waste = {Doc{}, Pay1{}, Pay1{Act1: d}, Pay2{},
       Pay2{Act2: d}, Pay3{}, Pay3{Act3: d},
       Pay4{}, Pay4{Act4: d}}
Pc2 = {}
Pc3 = {}
Pc4 = {}

Data at Locations: ...
```

Figure 6.14: Specification2: Analysis2 Results

Chapter 7

Conclusion

In the previous chapters we have presented a framework for doing insider analysis, and designed and programmed a tool that implement the elements of the framework. We evaluated our tool and found it to be a correct implementation of the Insider Framework.

7.1 Achievements

We have extended an existing theory [16] and focused on providing details that made that theory implementable. The extensions where mainly to add access-control to data and add a logging-component to the framework. After the extensions where in place, we programmed a tool for performing insider analysis. The tool implements three analysis described in the Insider Framework. Although the previous chapter contain a lot of theory we believe that our analysis and our tool could be helpful in a high-security, real-world system. Our tool should be easy to connect to a surveillance system, as the only thing that is needed for our system is a log file of actions performed in the system.

We have not heard about other similar tools and have thus not compared our tool to any other.

We believe that we have succeeded in developing an implementable framework for insider analysis and we are proud of the tool and the theory developed.

7.2 Limitations

Our tool has some limitations. There are a few integrity checks on the input files, especially the process definitions and the log files. The tool could have handled errors in the process definitions and log files more gracefully. The user interface could also display the result of the analyses in a more "graphical" way.

Analysis2 seems to be unrealistic as it is too pessimistic when input and output actions are not logged. It assumes the worst and actors give away all their data, witch is not very realistic in all cases.

7.3 Future Work

We have performed one iteration in the process of developing framework for doing insider analysis. The framework can be extended endlessly to make it more realistic and more flexible. We will list a few extensions that came to your mind while working on the project.

- **More complex tuples:** We chose to work with simple tuples, the theory and the tool could easily be extended with more complex tuple structures.
- **Recursive process definitions:** We did not allow process definitions to be recursive, as the syntax of insCalc did not provide support for specifying process definitions. By extending the syntax of insCalc to allow process definitions the invocation of process variables could be recursive.
- **Encrypt data with multiple keys:** We did not allow data to be encrypted with more than one key. The abstract system could be extended to allow data to be encrypted multiple times.
- **Locations that can be locked:** It could be nice to have locations that actors can lock, and thus make it impossible for other actors to enter the location while it was locked.
- **Movable locations:** Some locations could be marked as movable locations, that actors could move from one place to another. This would be

useful in modeling cars, elevators, and other movable objects in the real-world.

- **Probability on events:** It could give a better result to add probabilities on the likelihood that e.g., an actor gives another actor some data. The analysis would then take this into account and produce a result that depended on those probabilistic values.

Appendix A

Test Systems

A.1 Specification1

A.1.1 Test1.spe

```
locations: Hall{*:m_,o,r,i}(phys),
    Room1{*:m}(phys),Room2{*:m}(phys),
    Room3{*:m}(phys),Room4{*:m_}(phys),
    Room5{key2:m_}(phys),Room6{Doc:m_}(phys),
    Kitchen{Hall:m_}(phys),
    Waste{*:o,i,r}(phys),
    Pc1{*:e,o,i,r}(dig),
    Printer{*:o,i,r}(dig);
connections:Hall->Room1, Room1->Hall,
        Hall->Room2, Room2->Hall,
        Hall->Room3, Room3->Hall,
        Hall->Room4, Room4->Hall,
        Hall->Room5, Room5->Hall,
        Hall->Room6, Room6->Hall,
        Hall->Kitchen, Kitchen->Hall,
        Kitchen->Waste, Room1->Pc1,
        Room2->Printer, Pc1->Printer,
        Printer->Pc1;
actors: Act1@Room1, Act2@Room2; data: Doc{Room4:d}@Waste,
key1{}@Act1, key2{key1:d}@Act1,
    Pin{}@Room5;
```

A.1.2 Test1.pde

```
Act1 := move("Hall").move("Kitchen").
        in(!doc)@"Waste".move("Hall").move("Room4").
        decrypt(doc, !doc_decrypted).move("Hall").
        move("Room6").nil
```

A.1.3 Test1.log

```
(0, Actor(Act1), Room1, Hall, m);
(1, Location(Hall), Hall, Kitchen, m);
(2, Actor(Act1), Kitchen, Hall, m);
(3, Location(Hall), Hall, Room4, m);
(4, Actor(Act1), Room4, Hall, m);
(4, Key(Doc), Hall, Room6, m)
```

A.2 Chain of Keys

A.2.1 Chain_Of_Keys.spe

```
locations:  HALL{*:m, o, r, i}(phys),
        ROOM1{key1:m,i,r}(phys),
        ROOM2{key2:m_,i,r}(phys),
        ROOM3{key3:m,i,r}(phys),
        ROOM4{key4:m_,i,r}(phys),
        ROOM5{key5:m,i,r}(phys),
        ROOM6{key6:m_,i,r}(phys);

connections: HALL->ROOM1, ROOM1->HALL,
        HALL->ROOM2, ROOM2->HALL,
        HALL->ROOM3, ROOM3->HALL,
        HALL->ROOM4, ROOM4->HALL,
        HALL->ROOM5, ROOM5->HALL,
        HALL->ROOM6, ROOM6->HALL;

actors: ACT1@HALL, ACT2@HALL;
data: key1{ACT1:d; ACT2:d}@HALL, key2{ROOM1:d}@ROOM1,
    key3{ROOM2:d}@ROOM2, key4{ROOM3:d}@ROOM3,
    key5{ROOM4:d}@ROOM4, key6{ROOM5:d}@ROOM5;
```

A.2.2 Chain_Of_Keys.pde

```
ACT1 := in(!key1)@"HALL".decrypt(key1,!dec_key1).
        move("ROOM1").in(!key2)@"ROOM1".
        decrypt(key2, !dec_key2).move("HALL").
      move("ROOM2").in(!key3)@"ROOM2".
      decrypt(key3,!dec_key3).move("HALL").
      move("ROOM3").in(!key4)@"ROOM3".
      decrypt(key4,!dec_key4).move("HALL").
      move("ROOM4").in(!key5)@"ROOM4".
      decrypt(key5,!dec_key5).move("HALL").
      move("ROOM5").in(!key6)@"ROOM5".
      decrypt(key6,!dec_key6).move("HALL").
      move("ROOM6").move("HALL").nil
```

A.2.3 Chain_Of_Keys.log

```
(1, Key(key2), HALL, ROOM2, m);
(2, Key(key4), HALL, ROOM4, m);
(3, Key(key6), HALL, ROOM6, m)
```

A.3 The Long Hall

A.3.1 The_Long_Hall.spe

```
locations: HALL{*:m_,o_,r_,i_}(phys),
    JAN{key1:m_; JAN:r,i,o_}(phys),
    OFF{1234:m_,o,i,r}(phys),
    ROOM1{*:m}(phys),ROOM2{*:m}(phys),
    ROOM3{*:m_}(phys),ROOM4{*:m}(phys),
    ROOM5{*:m}(phys),ROOM6{*:m}(phys),
    KITCHEN{HALL:m_}(phys), WASTE{*:o,i,r}(phys),
    PC1{*:e,o,i,r}(dig), PRINTER{*:o,i,r}(dig);
connections:HALL->JAN, JAN->HALL,
            HALL->OFF, OFF->HALL,
        HALL->ROOM1, ROOM1->HALL,
        HALL->ROOM2, ROOM2->HALL,
        HALL->ROOM3, ROOM3->HALL,
        HALL->ROOM4, ROOM4->HALL,
        HALL->ROOM5, ROOM5->HALL,
        HALL->ROOM6, ROOM6->HALL,
        HALL->KITCHEN, KITCHEN->HALL,
        KITCHEN->WASTE, ROOM1->PC1,
        ROOM2->PRINTER, PC1->PRINTER,
        PRINTER->PC1;
actors: USER@OFF, JANITOR@JAN, WORKER@ROOM1;
data: DOC{}@WASTE, key1{}@USER, key2{key1:d}@USER,
    PIN{}@JAN;
```

A.3.2 The_Long_Hall.pde

```
JANITOR:=encrypt("trash" ,{JANITOR:d}, !trash).
         move("HALL").move("KITCHEN").out(trash)@"WASTE".
         read(!waste)@"WASTE".move("HALL").move("JAN").nil;

USER:=encrypt("old_banana" ,{*:d}, !old_banana).
      move("HALL").move("KITCHEN").out(old_banana)@"WASTE".
      read(!waste)@"WASTE".move("HALL").move("ROOM2").
      read(!everything)@"PRINTER".move("HALL").move("OFF").nil;

WORKER:=out("document")@"PC1".in(!document)@"PC1".
eval("Printing", out(document)@"PRINTER".nil)@"PC1".nil
```

A.3.3 The_Long_Hall.log

```
(1, Actor(USER), OFF, HALL, m);
(2, Actor(JANITOR), JAN, HALL, m);
(3, Key(PIN), HALL, JAN, m);
```

```
(4, Location(JAN), JAN, JAN, o);
(5, Location(HALL), HALL, KITCHEN, m);
(6, Location(JAN), JAN, HALL, m);
```

A.4 The PayCheck

A.4.1 PayCheck.spe

```
locations: Hall{*:m_,o,r,i}(phys),
    Room1{*:m,i,o,r}(phys),Room2{*:m,i,o,r}(phys),
    Room3{*:m,i,o,r}(phys),Room4{*:m,i,o,r}(phys),
    Wc{*:m,i,o,r}(phys),
    Kitchen{Hall:m,i,o,r}(phys),
    Waste{Kitchen:o,i,r}(phys),
    Pc1{Act1:e,o,i,r}(dig),
    Pc2{Act2:e,o,i,r}(dig),
    Pc3{Act3:e,o,i,r}(dig),
    Pc4{Act4:e,o,i,r}(dig);
connections:Hall->Room1, Room1->Hall,
        Hall->Room2, Room2->Hall,
        Hall->Room3, Room3->Hall,
        Hall->Room4, Room4->Hall,
        Hall->Wc, Wc->Hall,
        Hall->Kitchen, Kitchen->Hall,
        Kitchen->Waste, Room1->Pc1,
        Room2->Pc2, Room3->Pc3, Room4->Pc4;
actors: Act1@Room1, Act2@Room2, Act3@Room3, Act4@Room4;
data: Doc{}@Waste, Pay1{Act1:d}@Pc1, Pay2{Act2:d}@Pc2,
      Pay3{Act3:d}@Pc3, Pay4{Act4:d}@Pc4;
```

A.4.2 PayCheck.pde

```
Act1 := in(!pay1)@"Pc1".decrypt(pay1, !pay1_dec).
      move("Hall").move("Kitchen").nil;
Act2 := in(!pay2)@"Pc2".decrypt(pay2, !pay2_dec).
      move("Hall").move("Kitchen").nil;
Act3 := in(!pay3)@"Pc3".decrypt(pay3, !pay3_dec).
        move("Hall").move("Kitchen").nil;
Act4 := in(!pay4)@"Pc4".decrypt(pay4, !pay4_dec).
```

```
        move("Hall").move("Kitchen").nil
```

A.4.3 PayCheck.log

```
(0, Actor(Act4), Room4, Hall, m);
(1, Actor(Act3), Room3, Hall, m);
(2, Actor(Act2), Room2, Hall, m);
(3, Actor(Act1), Room1, Hall, m);
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] R. H. Anderson and R. C. Brackney. Understanding the insider threat. RAND Corporation, Santa Monica, CA, U.S.A.

[3] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge, 1998.

[4] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gianluigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The klaim project: Theory and practice. pages 1–51, 2005.

[5] Matt Bishop. The insider problem revisited. *Proceedings of the 2005 workshop on New security paradigms*, pages 75–76, 2005.

[6] Eric Brown. *Windows Forms in Action.* Manning, 2006.

[7] F# Developer Community. hubfs: The place for f#. `http://www.hubfs.net`.

[8] René Rydhof Hansen, Christian W. Probst, and Flemming Nielson. Sandboxing in myklaim. pages 1–8, 2005.

[9] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'reilly, 1995.

[10] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[11] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.

[12] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, 1999.

[13] Hanne Riis Nielson and Flemming Nielson. Flow logic, a multi-paradigmatic approach to static analysis. *The Essence of Computation*, 2566:223–244, 2002.

[14] SooHyoung Oh. Ocamllex tutorial. `http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamllex-tutorial/`.

[15] SooHyoung Oh. Ocamlyacc tutorial. `http://plus.kaist.ac.kr/~shoh/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial/`.

[16] Christian Probst, Rene R Hansen, and Flemming Nielson. Where can an insider attack? pages 1–17, 2006.

[17] Various. Ocaml tutorial. `http://www.ocaml-tutorial.org/`.