Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Report on Formal Analysis of Security Properties
D3.5.6

Due date of deliverable: 30/11/2007
Actual submission date: 1/11/2007

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: 3.5
Task number: 3.5.6

Responsible institution: Rutherford Appleton Laboratory,
Science & Technology Facilities Council,
Harwell Science and Innovation Campus,
Didcot, Oxon OX11 0QX, United Kingdom
Editor & and editor's address: Alvaro E. Arenas and Benjamin Aziz

Version 1.0 / Last edited by Benjamin Aziz / 04/12/07

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.0	08/08/07	Alvaro Arenas	STFC	First draft
0.1	08/10/07	Erica Yang	STFC	Added the first version of the revised authentication protocol
0.2	09/10/07	Benjamin Aziz	STFC	Updated the Mutual Authentication protocol
0.3	25/10/07	Benjamin Aziz	STFC	Completed the Section on the Verification of the Mutual Authentication protocol
0.4	26/10/07	Benjamin Aziz	STFC	Completed the Data Storage Confidentiality Goal/Anti-Goal Models
0.5	30/10/07	Benjamin Aziz	STFC	Completed the Data Storage Integrity and SSO Goal/Anti-Goal Models
0.6	01/11/07	Alvaro Arenas	STFC	Added Introduction, Introduction to Requirements and KAOS Summary, Conclusion
0.7	20/11/07	Benjamin Aziz	STFC	Applied Corrections Corresponding to the First Reviewer's Comments
0.8	21/11/07	Benjamin Aziz	STFC	Applied Corrections Corresponding to the Second Reviewer's Comments
0.9	03/12/07	Benjamin Aziz	STFC	Updated the Goal and Anti-Goal Models including the Figures
1.0	04/12/07	Benjamin Aziz	STFC	Corrected a few Typos

Reviewers:
Christine Morin and Alexander Reinefeld

Tasks related to this deliverable:

Task No.	Task description	Partners involved
T3.5.6	Formal Analysis of Security Properties	STFC*

*task leader

Contents

List of Figures

Executive Summary

This document is the first report of the task on formal analysis of security properties in XtreemOS. The report focuses on two key areas for the development of security-critical systems: security requirements and protocol verification. Each area is formally analysed following suitable methodologies and techniques.

For the security requirements part, we have used the KAOS goal-oriented requirements engineering methodology for analysing previously-defined security requirements of XtreemOS. This methodology has been tailored to model security aspects by including an anti-goal model, a variant of an *attack tree*. The root node of the tree is the global goal of an attacker. Children of a node are refinements of this goal, and leafs therefore represent attack steps. This representation offers support for a systematic exploration of the attacks on the system-to-be.

At the protocol level, we formally model one of the chief protocols in the XtreemOS security architecture: the *mutual-authentication protocol*. The protocol is modelled using the applied π-calculus; the ProVerif tool is used to verify automatically that the mutual-authentication property holds. An attacker is also included in the model, and model-checking techniques are applied to verify that data leakage does not occur in the presence of such an attacker.

The task of formal analysis revealed a number of outcomes for both the requirement modelling and the protocol verification parts. For the requirement modelling part, the possibility of a number of attacks related to breaches of confidentiality, integrity and single-sign-on properties were revealed at the requirement level, such as false ownerships, bad credentials, masquerading, misjudgment of data origin and multiple ids per user. On the other hand, the protocol verification part confirmed the minimum secrecy and authenticity properties of the proposed protocol, even though these were judged to be dependent on the size of the Diffie-Hellman numbers and the existence of secure, non-cryptographic, channels.

1 Introduction

Formal methods are mathematically-based techniques for the specification, development and verification of software and hardware systems. The use of formal methods for software and hardware design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analyses can contribute to the reliability and robustness of a design.

Formal methods can help us articulate precisely a system's boundary, i.e. the interface between the system and its environment; characterize precisely a system's behaviour; define precisely a system's desired properties; prove a system meets its specification; and determine under what circumstances a system does not meet its specification. For example, some methods produce counter-examples, such as intruder scenarios, which explain why the system is flawed.

These capabilities of formal methods help the practitioner in two ways:

- *Through specification*, focusing the system designer's attention. What is the interface? What are one's assumptions about the system's environment? What is the system supposed to do under this condition or that condition? What happens if that condition is not met? What are the system's invariant properties?

- *Through verification*, providing additional assurance. Relying on a proof that a system meets its security goals is better than relying on a gut feeling.

In this document we report the application of formal methods to two important parts of the development of the security services in XtreemOS: security requirements and protocol verification.

First, we apply a formally-based requirements-engineering methodology to define and analyse three important security requirements of XtreemOS identified previously in deliverable D3.5.2: data storage confidentiality, data storage integrity and single sign-on. Each of these requirements has been modelled following the KAOS goal-oriented methodology, including attack scenarios that allow one to determine potential vulnerabilities in the system. This work helps designers to describe unambiguously their requirements, to define the assumptions of the system environment, and to specify XtreemOS security services and their interface.

Second, we verify formally the XtreemOS mutual-authentication protocol, presented previously in deliverable D3.5.3. This protocol aims at proving a user's identity in the context of a Virtual Organisation (VO), based on the classic Diffie-Hellman key agreement protocol. Our approach has consisted in (1) defining formally the meaning of the mutual authentication property; then (2) modelling the protocol using the applied π-calculus, a variant of the well-known π-calculus; and

(3) verifying automatically that the mutual-authentication property holds in the protocol using the ProVerif tool. We have also verified that the protocol does not leak data to external attackers. The initial version of the protocol proved to be correct, but some parts of the protocol could be interpreted ambiguously, so as a result of this formal work a new version of the protocol has been produced.

2 Formal Modelling of Security Requirements

2.1 Introduction

The elaboration of requirements is a crucial step in the development of software-intensive security-critical applications. A missing, inadequate, imprecise or inconsistent requirement might cause the resulting application to have vulnerabilities, despite the huge amount of work that might be invested in implementing the elaborated security requirements.

Security requirements are generally organised into the following non-exhaustive taxonomy known as the C.I.A. taxonomy [9]:

- **Confidentiality requirements** prescribe that sensitive information be not disclosed to unauthorized agents.
- **Integrity requirements** prescribe that information is changed only in a specified and authorized manner.
- **Availability requirements** prescribe that the system must be accessible and usable upon demand by some authorized entity.

This chapter models formally a subset of the security requirements identified previously in XtreemOS deliverable D3.5.2, *Security Requirements for a Grid-Based OS* [1]. Above taxonomy has motivated us to select three security-critical requirements identified in D3.5.2: data storage confidentiality, data storage integrity, and single sign-on. It could be argued that this selection is not exhaustive, but we hope that the method described below could guide developers to analyse their security requirements more rigorously according to the project needs.

It is not yet very clear how to identify security requirements systematically through the various stages of the requirements engineering process. Security requirements are sometimes left unspecified, notably because of the cost of identifying them and because they are not clearly visible in the implementation, unlike functional requirements. The oldest practices rely on general guidelines, partial reuse of existing requirements model, or peer reviewing. Two other trends have emerged, the first one relies on refinement of the desired security properties and the second one relies on attack identification and prevention.

For the formal work on security requirements in XtreemOS, we have followed the recent trends on requirements engineering — refinement of security properties and attack identification — by applying the KAOS requirements-engineering methodology [20].

2.2 Goal-Oriented Requirements Engineering

KAOS is a generic methodology that is based on the capturing, structuring and precise formulation of the system goals [20]. A goal is prescriptive description of system properties, formulated in non-operational terms. A system includes not only the design but also its environment. Goals are refined and operationalised in a top-down manner as the system is designed or bottom up approach while re-engineering existing systems. The approach also supports adverse environments, composed of possibly malicious external agents trying to violate the system goal rather than to collaborate in the goal fulfillment. As a Grid system is typically composed of a number of interacting nodes immersed in an open and possibly adverse environment, this approach fits our needs well.

A KAOS model is composed of a number of interrelated sub-models:

- The *goal model* captures and structures the assumed and required properties of a system. In our case we will focus on security properties.
- The *agent model* assigns goals to agents in a realizable way. Discovering the responsible agents is the criterion to stop a goal-refinement process.
- The *object model* is used to identify the concepts of the application domain that are relevant with respect to the requirements and to provide static constraints on the operational systems that will satisfy the requirements. The object model consists of objects from the stakeholders' domain and objects introduced to express requirements or constraints on the operational system.
- The *operation model* details, at state-transition level, the actions an agent has to perform to reach the goals it is responsible for.
- The *anti-goal model* captures attacks on the system and how they are addressed. This model is built in parallel with the goal-model and helps discover goals that will improve the robustness of the system, especially against malicious agents whose aim is to break the high level goals of the system.

In our work, we have focused only on the goal and anti-goal models, since they allow one to refine security properties and to construct attack trees respectively.

2.2.1 The Goal Model

The goal model is the driving model of the KAOS language. It declares the goals of the composite system. A goal defines an objective the composite system should meet, usually through the cooperation of multiple agents. An example of goal for

a meeting scheduling problem is the goal **Goal** [Maintain Data Storage Confidentiality] requiring that the confidentiality of a data storage must be maintained all time.

Goals are organized in AND/OR refinement-abstraction hierarchies where higher-level goals are in general strategic, coarse-grained and involve multiple agents whereas lower-level goals are in general technical, fine-grained and involve less agents. In such structures, AND-refinement links relate a goal to a set of subgoals (called refinement) possibly conjoined with domain properties; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. OR-refinement links may relate a goal to a set of alternative refinements; this means that satisfying one of the refinements is a sufficient condition in the domain for satisfying the goal. Goal refinement ends when every subgoal is realizable by some individual agent assigned to it, that is, expressible in terms of conditions that are monitorable and controllable by the agent. A *requirement* is a terminal goal under responsibility of an agent in the software-to-be; an *expectation* is a terminal goal under responsibility of an agent in the environment (unlike requirements, expectations cannot be enforced by the software-to-be). Goals prescribe intended behaviors; they can be formalized in a real-time temporal logic.

2.2.2 The Anti-Goal Model

In the context of security engineering, it is important to model attackers, their goals and capabilities, the software vulnerabilities they can monitor or control, and attacks that satisfy their goals based on their capabilities and on the system's vulnerabilities. In [21], such models are called *anti-models* and the attacker's goals are called *anti-goals*, including malicious obstacles to security goals. Anti-goals should of course be distinguished from the goals the system under consideration should satisfy. Anti-models should lead to the generation of more subtle threats and the derivation of more robust security requirements as anticipated countermeasures to such threats.

2.2.3 Linear Temporal Logic

Both goals and anti-goals are formally modelled using Linear Temporal Logic (LTL) formulae. *Formulae*, P, Q, in LTL are built from the usual logic connectors $(\wedge \vee \neg \rightarrow \leftrightarrow)$ as well as the following temporal modal operators:

- $\circ P$: This is the *next* operator, which says that P has to hold in the next state.
- $\Box P$: This is the *always* operator, which says that P has to hold from this point in time and all subsequent states.

- $\Diamond P$: This is the *eventually* operator, which says that P has to hold at some time in the future.
- $P \; \mathcal{U} \; Q$: This is the *until* operator, which says that Q has to hold now or some time in the future and until then, P must hold. Once Q occurs, P does not have to hold any more.

We also write, $(P \Rightarrow Q)$ to mean $\Box(P \rightarrow Q)$. In the following sections, we shall use the language of LTL in modelling the security goal and anti-goal models.

2.3 Modelling Security Requirements

In the following sections, we define the goal and anti-goal models for three security properties: data storage confidentiality, integrity and single-sign on.

2.3.1 Data Storage Confidentiality Goal Model

The confidentiality requirement for data storage we consider here is one of the security requirements recognised by the test case scenarios in the XtreemOS project. The requirement is highlighted as R78 in deliverable D3.5.2 [1].

The requirement considers only *access control* as a mechanism for ensuring the confidentiality of data stored on a system. Therefore, we do not consider other mechanisms for achieving confidentiality such as encryption or non-interference here. Only valid *principals* of the Grid can access *data* stored on resources. *Principals* are defined as being the users, administrators or services present in the Grid, whereas *data* is defined as being any data stored on the local or grid-based filesystem, data present in a shared memory or data contained within a license. A valid principal then is defined as follows:

- the owner of the data, or
- a non-owner principal who is a member of a VO, and:
  - the VO is authorised to access the data, and
  - the principal is assigned to a task requiring access to the data.

The last condition assumes the principle of *least privilege*, which states that principals should obtain no more than their minimum (access) rights necessary to achieve their functionality. Any principal not satisfying the conditions above is considered to be an *invalid principal*.

Using KAOS concepts, the goal model of the data storage confidentiality is shown in Figure 1. Before defining the formal goal model for confidentiality, we introduce a few useful sets and predicates:

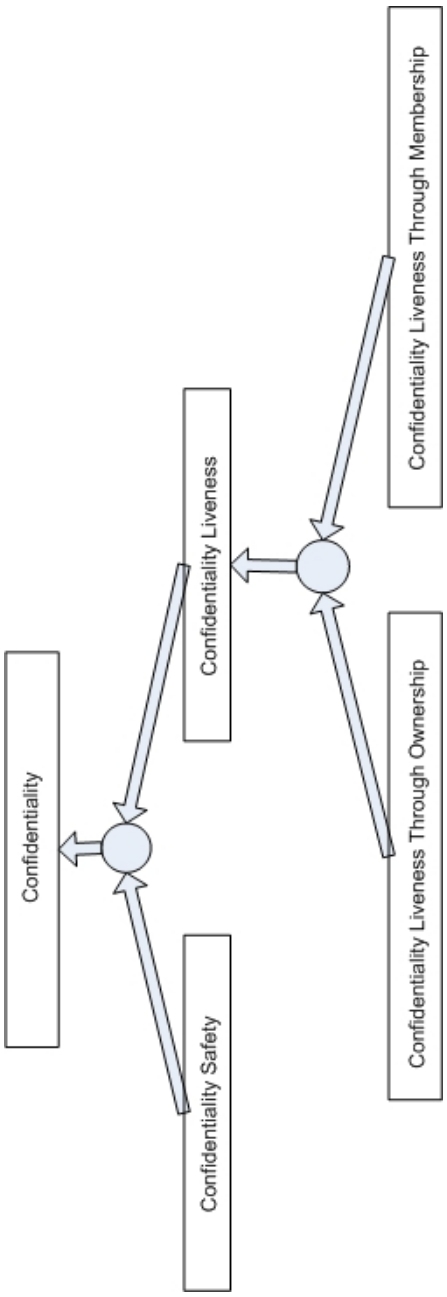

Figure 1: The goal model for data storage confidentiality.

- *VO*: the set of all VOs.
- *Data*: the set of all data including that which may be created and used by XtreemOS applications.
- *Principal*: the set of principals in a VO. Principals may be users, administrators or services present on the Grid.
- *Task*: the set of tasks running in a VO. Tasks are usually assigned to principals in the VO.
- *Entity*: is the union set $VO \cup Data \cup Task$.
- *Credential*: This is the set of credentials. A credential may be a password, a certificate or any other mechanism.
- *Attribute*: This is the set of attributes of a credential. This set includes data ownership, VO membership, task assignment and data read and write attributes.
- *Right*: is the set of access rights, *read* and *write*. Note that $Right \subset Attribute$. For the case of the confidentiality property, we only consider the right to *read* data.
- *owner* : $Principal \times Data \rightarrow \mathbb{B}$ is a predicate on principals and datasets denoting that a principal owns a dataset.
- *member* : $Principal \times VO \rightarrow \mathbb{B}$ is a predicate on principals and VOs denoting that a principal is a member of a VO.
- *assigned* : $Principal \times Task \rightarrow \mathbb{B}$ is a predicate denoting that a principal is assigned to some task.
- *requires* : $Task \times Data \rightarrow \mathbb{B}$ is a predicate on tasks and datasets denoting that a task requires a dataset.
- *hasVORights* : $Principal \times Data \times VO \times Right \rightarrow \mathbb{B}$ is a predicate on principals and datasets denoting that a principal has a right (read or write) to access the dataset and that the right was issued by some VO.
- *binding* : $Principal \times Entity \times Attribute \times Credential \rightarrow \mathbb{B}$ is a predicate on principals and entities denoting that a principal is bound to the entity by means of some credential and this binding has the specified attribute.
- *issued_By* : $Credential \times VO \rightarrow \mathbb{B}$ is a predicate stating that a credential is issued by a VO.

Using these sets and predicates, we formalise the top-level goal of data storage confidentiality as follows:

Goal [Confidentiality]
FormalDef $\forall p \in \textit{Principal}, d \in \textit{Data}$:
$\textit{authorised}(p, d, \textit{read}) \Leftrightarrow$
$(\textit{owner}(p, d) \vee (\exists vo \in \textit{VO}, t \in \textit{Task} : \textit{member}(p, vo) \wedge \textit{hasVORights}(p, d, vo, \textit{read}) \wedge$
$\textit{assigned}(p, t) \wedge \textit{requires}(t, d)))$

where $\textit{authorised} : \textit{Principal} \times \textit{Data} \times \textit{Right} \rightarrow \mathbb{B}$ is a predicate denoting that a principal is authorised to access some data with some right (in our case, the right to read the data). Next, we break down the top-level goal in terms of the following two subgoals:

Goal [Confidentiality Safety]
FormalDef $\forall p \in \textit{Principal}, d \in \textit{Data}, r \in \textit{Right}$:
$\textit{authorised}(p, d, \textit{read}) \Rightarrow$
$(\textit{owner}(p, d) \vee (\exists vo \in \textit{VO}, t \in \textit{Task} : \textit{member}(p, vo) \wedge \textit{hasVORights}(p, d, vo, \textit{read}) \wedge$
$\textit{assigned}(p, t) \wedge \textit{requires}(t, d)))$

Goal [Confidentiality Liveness]
FormalDef $\forall p \in \textit{Principal}, d \in \textit{Data}$:
$\textit{authorised}(p, d, \textit{read}) \Leftarrow$
$(\textit{owner}(p, d) \vee (\exists vo \in \textit{VO}, t \in \textit{Task} : \textit{member}(p, vo) \wedge \textit{hasVORights}(p, d, vo, \textit{read}) \wedge$
$\textit{assigned}(p, t) \wedge \textit{requires}(t, d)))$

The first subgoal can only be true if it is true that the principal is either the owner or a valid member of a VO. Therefore, it denotes safety of confidentiality through the sufficiency of the right side condition. In the second subgoal, the left side must be true if the principal is the owner of the data or a valid VO member for the goal to be true. Therefore, it denotes liveness of confidentiality. Now, we break down the second subgoal further on to the following two subgoals:

Goal [Confidentiality Liveness Through Ownership]
FormalDef $\forall p \in \textit{Principal}, d \in \textit{Data}, r \in \textit{Right}$:
$\textit{authorised}(p, d, \textit{read}) \Leftarrow \textit{owner}(p, d)$

Goal [Confidentiality Liveness Through Membership]
FormalDef $\forall p \in \textit{Principal}, d \in \textit{Data}, r \in \textit{Right}$:
$\textit{authorised}(p, d, \textit{read}) \Leftarrow (\exists vo \in \textit{VO}, t \in \textit{Task} : \textit{member}(p, vo) \wedge$
$\textit{hasVORights}(p, d, vo, \textit{read}) \wedge \textit{assigned}(p, t) \wedge \textit{requires}(t, d))$

Note that *read* in the first subgoal does not affect the predicate since an owner has exclusive rights over its datasets (including the right to read the data). Finally,

we give Grid-based domain-specific definitions of the predicates appearing on the right side of these implications.

Definition [Owner Credential Validated]
FormalDef $(\forall p \in \textit{Principal}, d \in \textit{Data}) : \ \textit{owner}(p, d) \Leftrightarrow$
$(\exists cr \in \textit{Credential} : \ \textit{binding}(p, d, \textit{own}, cr) \ \wedge \ \textit{well\_defined}(cr))$

Definition [Assignment Credential Validated]
FormalDef $(\forall p \in \textit{Principal}, t \in \textit{Task}) : \ \textit{assigned}(p, t) \Leftrightarrow$
$(\exists cr \in \textit{Credential} : \ \textit{binding}(p, t, \textit{assigned}, cr))$

Definition [Member Credential Validated]
FormalDef $(\forall p \in \textit{Principal}, vo \in \textit{VO}) : \ \textit{member}(p, vo) \Leftrightarrow$
$(\exists cr \in \textit{Credential} : \ \textit{binding}(p, vo, \textit{member}, cr))$

Definition [Rights Credential Validated]
FormalDef $(\forall p \in \textit{Principal}, d \in \textit{Data}, vo \in \textit{VO}, r \in \textit{Right}) :$
$\textit{hasVORights}(p, d, vo, \textit{read}) \Leftrightarrow (\exists cr \in \textit{Credential} : \ \textit{binding}(p, d, \textit{read}, cr) \wedge$
$\textit{issued\_By}(cr, vo))$

In addition to the above definitions, we state the well-definedness of ownership credentials as follows:

Definition [Credential Well-Definedness]
FormalDef $\forall cr \in \textit{Credential} :$
$\textit{well\_defined}(cr) \ \Leftrightarrow \ (\forall p, p' \in \textit{Principal}, d \in \textit{Data} :$
$\textit{binding}(p, d, \textit{own}, cr) \ \wedge \ \textit{binding}(p', d, \textit{own}, cr) \ \Rightarrow \ p = p')$

The requirement essentially states that a credential cannot refer to more than one principal. An example of this requirement is that a certificate, which has a unique serial number, must refer to only one principal. The public key of the principal contained in the certificate represents the data to which the principal is bound.

2.3.2 Data Storage Confidentiality Anti-Goal Model

The anti-goal model of data storage confidentiality is shown in Figure 2. As mentioned in section 2.2, the anti-goal model can be obtained by negating the goal model at all levels. This will help reveal internal vulnerabilities and external attacks targeted against the data storage confidentiality property. We start at the top-level:

To obtain the formal model, we start by negating the "Confidentiality Safety",

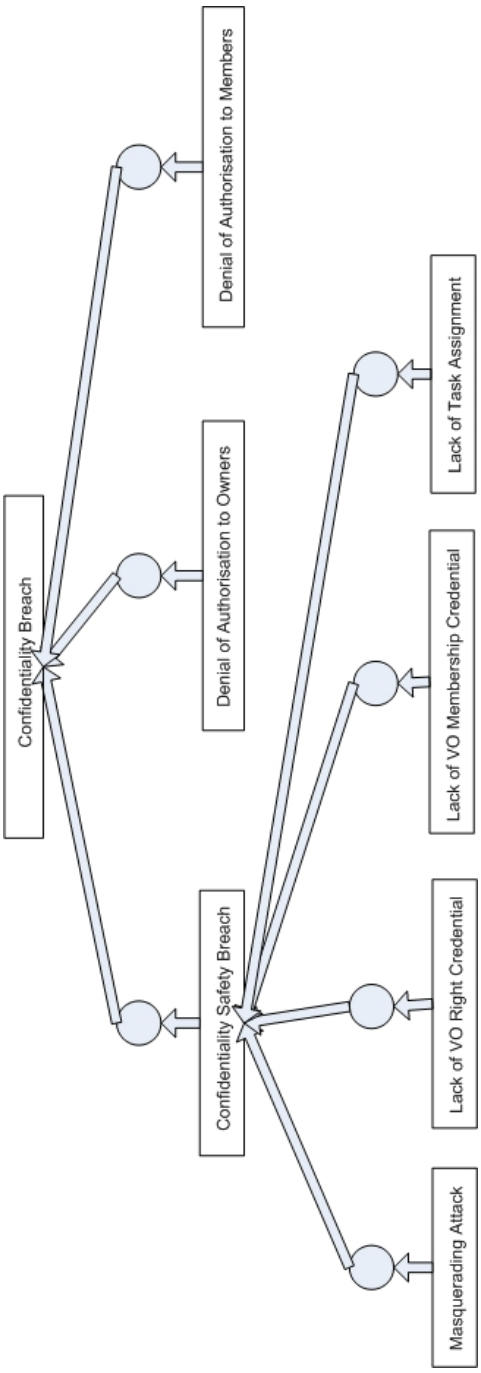

Figure 2: The anti-goal model for data storage confidentiality.

the "Confidentiality Liveness Through Ownership" and the "Confidentiality Liveness Through Membership" subgoals, as follows:

AntiGoal [Confidentiality Safety Breach]
FormalDef $\exists p \in \textit{Principal}, d \in \textit{Data}: \ \Diamond\,(\textit{authorised}(p, d, \textit{read}) \wedge$
$\neg(\textit{owner}(p, d) \ \vee \ (\exists vo \in \textit{VO}, t \in \textit{Task}: \ \textit{member}(p, vo) \wedge$
$\textit{hasVORights}(p, d, vo, \textit{read}) \wedge$
$\textit{assigned}(p, t) \ \wedge \ \textit{requires}(t, d))))$

AntiGoal [Denial of Authorisation to Owners]
FormalDef $\exists p \in \textit{Principal}, d \in \textit{Data}:$
$\Diamond\,(\neg\textit{authorised}(p, d, \textit{read}) \ \wedge \ \textit{owner}(p, d))$

AntiGoal [Denial of Authorisation to Members]
FormalDef $\exists p \in \textit{Principal}, d \in \textit{Data}:$
$\Diamond\,(\neg\textit{authorised}(p, d, \textit{read}) \ \wedge \ (\exists vo \in \textit{VO}, t \in \textit{Task}: \ \textit{member}(p, vo) \wedge$
$\textit{hasVORights}(p, d, vo, \textit{read}) \wedge$
$\textit{assigned}(p, t) \ \wedge \ \textit{requires}(t, d)))$

Though interesting, the last two anti-subgoals are outside the scope of confidentiality breaches since these deal more with the denial of service-like vulnerabilities. Therefore, we only focus on the first anti-subgoal. It is possible to expand the "Confidentiality Safety Breach" anti-subgoal by substituting the definitions of the different predicates into the anti-subgoal. However, for lack of space, we shall consider only the interesting cases of the ownership and the VO rights predicates. We start with the former:

AntiGoal [Confidentiality Safety Breach]
FormalDef $\exists p \in \textit{Principal}, d \in \textit{Data}, \forall cr \in \textit{Credential}, vo \in \textit{VO}, t \in \textit{Task}:$
$\Diamond\,(\textit{authorised}(p, d, \textit{read}) \wedge$
$(\neg\textit{binding}(p, d, \textit{own}, cr) \ \vee \ \neg\textit{well\_defined}(cr)) \wedge$
$(\neg\textit{member}(p, vo) \ \vee \ \neg\textit{hasVORights}(p, d, vo, \textit{read}) \vee$
$\neg\textit{assigned}(p, t) \ \vee \ \neg\textit{requires}(t, d)))$

This anti-subgoal introduces the predicate, $\neg\textit{well\_defined}(cr)$, which expresses badly-defined credentials. By substituting the definition of this predicate into the anti-subgoal, we get.

AntiGoal [Masquerading Attack]
FormalDef $\exists p, p' \in \textit{Principal}, d \in \textit{Data}, \forall cr \in \textit{Credential}, vo \in \textit{VO}, t \in \textit{Task}:$
$\Diamond\,(\textit{authorised}(p, d, \textit{read}) \wedge$
$(\neg\textit{binding}(p, d, \textit{own}, cr) \vee$
$(\textit{binding}(p, d, \textit{own}, cr) \ \wedge \ \textit{binding}(p', d, \textit{own}, cr) \wedge$
$\neg(p = p'))) \wedge$

$(\neg \textit{member}(p, vo) \ \vee \ \neg \textit{hasVORights}(p, d, vo, \textit{read}) \vee$
$\neg \textit{assigned}(p, t) \ \vee \ \neg \textit{requires}(t, d)))$

This anti-subgoal models the case in which a credential refers to two different principals both as owners of the same dataset. In our opinion, this constitutes a form of masquerading attacks in which one principal pretends to be another by presenting information to the system pertaining to be the other principal.

On the other hand, expanding with the definition of the lack of Vo rights, $\neg \textit{hasVORights}(p, d, vo, \textit{read})$, yields the following anti-subgoal:

AntiGoal [Lack of VO Right Credential]
FormalDef $\exists p \in \textit{Principal}, d \in \textit{Data}, \forall cr \in \textit{Credential}, vo \in \textit{VO}, t \in \textit{Task}$:
$\Diamond \ (\textit{authorised}(p, d, \textit{read}) \wedge$
$(\neg \textit{owner}(p, d)) \wedge$
$(\neg \textit{member}(p, vo) \vee$
$(\neg \textit{binding}(p, d, \textit{read}, cr) \ \vee \ \neg \textit{issued\_By}(cr, vo)) \vee$
$\neg \textit{assigned}(p, t) \ \vee \ \neg \textit{requires}(t, d)))$

This anti-subgoal expresses a hacking attack in which the principal is authorised to read the data even though it has no VO rights credential. Similarly, it is possible to express other scenarios by expanding on the definitions of the $\neg$*assigned* and $\neg$*member* predicates, as follows:

AntiGoal [Lack of VO Membership Credential]
FormalDef $\exists p \in \textit{Principal}, d \in \textit{Data}, \forall cr \in \textit{Credential}, vo \in \textit{VO}, t \in \textit{Task}$:
$\Diamond \ (\textit{authorised}(p, d, \textit{read}) \ \wedge \ \neg \textit{owner}(p, d) \wedge$
$(\neg \textit{binding}(p, vo, \textit{member}, cr) \ \vee \ \neg \textit{hasVORights}(p, d, vo, \textit{read}) \vee$
$\neg \textit{assigned}(p, t) \ \vee \ \neg \textit{requires}(t, d)))$

AntiGoal [Lack of Task Assignment]
FormalDef $\exists p \in \textit{Principal}, d \in \textit{Data}, \forall cr \in \textit{Credential}, vo \in \textit{VO}, t \in \textit{Task}$:
$\Diamond \ (\textit{authorised}(p, d, \textit{read}) \ \wedge \ \neg \textit{owner}(p, d) \wedge$
$(\neg \textit{member}(p, vo) \ \vee \ \neg \textit{hasVORights}(p, d, vo, \textit{read}) \vee$
$\neg \textit{binding}(p, t, \textit{assigned}, cr) \ \vee \ \neg \textit{requires}(t, d)))$

Which express the two scenarios where the principal can read the data even though it is not a member of the VO nor is assigned to a task requiring the data.

2.3.3 Data Storage Integrity Goal Model

The second security requirement considered here is that of data storage integrity. This requirement appears as R80 in [1]. In this requirement, integrity is defined as the "inability to prevent illegal changes to data", both locally and remotely. Essentially, R80 identifies three scenarios in which data integrity is an issue:

- Data belonging to a principal may be stored in a remote trust domain. This has the implication that the domain may be able to modify the data without the principal's consent.
- Data belonging to a principal and stored in the local principal's trust domain may be shared with other external principals who may have access to the data. This implies that those principals may be able to modify the data, again without the principal's consent.
- Finally, a principal sourcing some data must make sure that the data integrity is validated. This is due to the fact that data may be sourced from non-owner principals who may have illegally modified the data.

In all these scenarios, a principal is authorised to modify the data if the principal is the owner of the data or a member of a VO and have the appropriate rights to make changes to the data.

Using the KAOS goal model, we can illustrate the data storage integrity requirement as in Figure 3. More formally, we can state the property in terms of temporal logic operators as follows:

Goal [Integrity]
FormalDef $\forall p \in \textit{Principal}, d \in \textit{Data}$:
$\Box\ ((\textit{authorised}(p, d, \textit{write}) \leftrightarrow$
$(\textit{owner}(p, d) \lor (\exists vo \in \textit{VO} :\ \textit{member}(p, vo) \land \textit{hasVORights}(p, d, vo, \textit{write})))) \land$
$(\textit{sourced}(p, d) \Rightarrow \textit{validated}(p, d)))$

where we introduce two new predicates, $\textit{sourced} : \textit{Principal} \times \textit{Data} \rightarrow \mathbb{B}$, which states that a principal has sourced some data and $\textit{validated} : \textit{Principal} \times \textit{Data} \rightarrow \mathbb{B}$ to state that a principal has validated the origin of the data. The integrity goal is further refined to the following subgoals:

Goal [Safety of Writing Integrity]
FormalDef $\forall p \in \textit{Principal}, d \in \textit{Data}$:
$\textit{authorised}(p, d, \textit{write}) \Rightarrow$
$(\textit{owner}(p, d) \lor (\exists vo \in \textit{VO} :\ \textit{member}(p, vo) \land \textit{hasVORights}(p, d, vo, \textit{write})))$

Goal [Liveness of Writing Integrity]
FormalDef $\forall p \in \textit{Principal}, d \in \textit{Data}$:
$\textit{authorised}(p, d, \textit{write}) \Leftarrow$
$(\textit{owner}(p, d) \lor (\exists vo \in \textit{VO} :\ \textit{member}(p, vo) \land \textit{hasVORights}(p, d, vo, \textit{write})))$

Goal [Sourcing Integrity]
FormalDef $\forall p \in \textit{Principal}, d \in \textit{Data}$:
$\textit{sourced}(p, d) \Rightarrow \textit{validated}(p, d)$

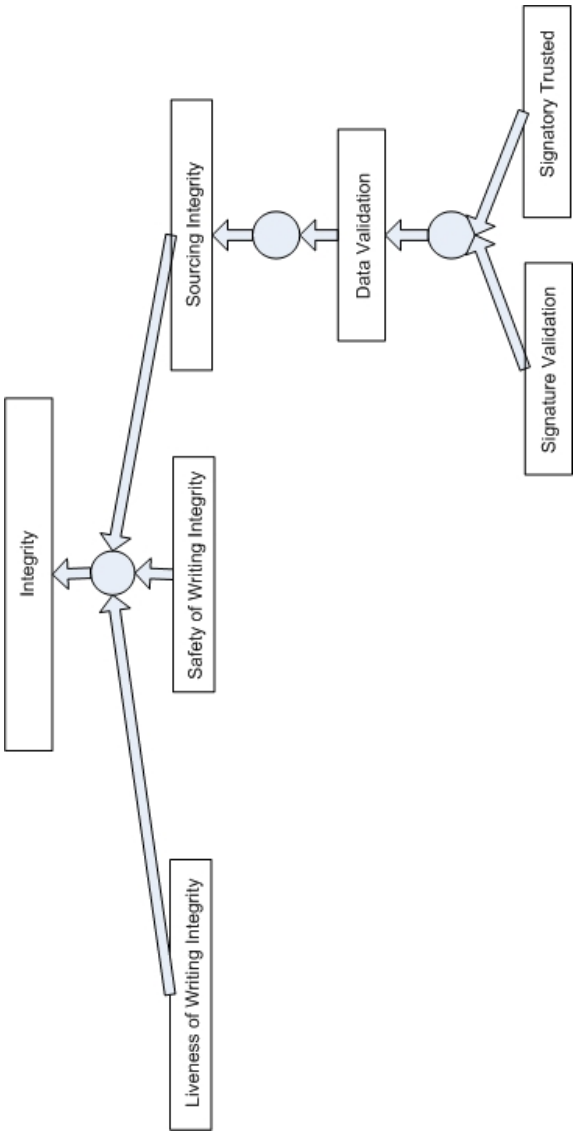

Figure 3: The goal model for data storage integrity.

We only consider the first and third subgoals here, since these are more related to the safety of our integrity property. The first subgoal states that in order to be able to write to some data, a principal must be either the owner of the data or a member of a VO with appropriate writing access right issued by the VO. Note here that in this subgoal, we are using predicates introduced in the previous section on confidentiality, except now we talk about writing rather reading access rights. In the third subgoal, every data that is sourced must be also validated by the principal that is sourcing it. The second goal is more related to scenarios where being the owner or valid VO member guarantees the principal the right to write to the dataset. We consider such goal as a liveness property. Next, we need to clarify by what we mean by data validation through the following subgoal:

Goal [Data Validation]
FormalDef $\forall d \in \mathit{Data}, p \in \mathit{Principal}$:
$\mathit{validated}(p, d) \Rightarrow (\exists p' \in \mathit{Principal} : \mathit{signed}(d, p') \wedge \mathit{trusts}(p, p'))$

The $\mathit{signed} : \mathit{Data} \times \mathit{Principal} \rightarrow \mathbb{B}$ predicate states that some data is signed by a principal and the $\mathit{trusts} : (\mathit{Principal} \cup \mathit{CA}) \times (\mathit{Principal} \cup \mathit{CA}) \rightarrow \mathbb{B}$ predicate states that a principal or a Certification Authority (CA) trusts another principal or CA. The definition of the *signed* and *trusts* predicates are given as follows:

Definition [Signature Validation]
FormalDef $\forall d \in \mathit{Data}, p \in \mathit{Principal}$:
$\mathit{signed}(d, p) \Leftrightarrow (\exists sig \in \mathit{Signature} : \mathit{keyBy}(sig, p) \wedge \mathit{hashBy}(sig, d))$

Definition [Signatory Trusted]
FormalDef $\forall p, p' \in \mathit{Principal} : \mathit{trusts}(p, p') \Leftrightarrow (\exists ca \in \mathit{CA} : \mathit{trusts}(p, ca) \wedge \mathit{trusts}(ca, p'))$

Where the first definition states that for some data to be signed by a principal, there must exist a digital signature such that the private key signing the data belongs to the principal and the hash in the signature belongs to the data. The second definition states that trust is a transitive relation.

2.3.4 Data Storage Integrity Anti-Goal Model

The anti-goal model for data storage integrity is defined by taking the negation of each of the goals and requirements defined in the previous section. Informally, the goal is illustrated in Figure 4. We start by negating the "Safety of Writing Integrity" and the "Sourcing Integrity" subgoals as follows:

AntiGoal [No Safety of Writing Integrity]
FormalDef $\exists p \in \mathit{Principal}, d \in \mathit{Data}$:

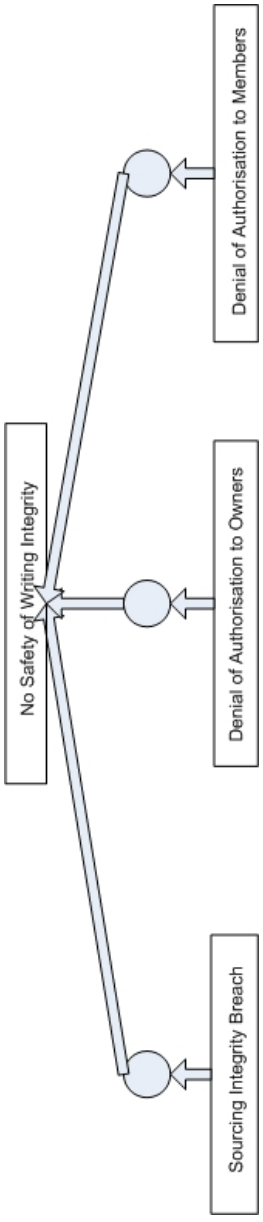

Figure 4: The anti-goal model for data storage integrity.

authorised$(p, d, write) \wedge$
$\neg$*owner*$(p, d) \ \wedge \ (\forall vo \in$ *VO* $: \ \neg$*member*$(p, vo) \ \vee \ \neg$*hasVORights*$(p, d, vo, write))$

AntiGoal [Sourcing Integrity Breach]
FormalDef $\exists p \in$ *Principal*, $d \in$ *Data* :
sourced$(p, d) \wedge \neg$*validated*(p, d)

Expanding along the first anti-subgoal, we could obtain the scenarios where the principal has managed to write to the dataset without being its owner nor being a valid member of a VO with appropriate rights. This anti-subgoal can be further expanded using credential-specific definitions and the credential well-definedness property. On the other, expanding the second anti-subgoal along the definition of *validated*(p, d) yields the following:

AntiGoal [Sourcing Integrity Breach]
FormalDef $\exists p \in$ *Principal*, $d \in$ *Data* :
sourced$(p, d) \wedge \neg(\forall p' \in$ *Principal* $: \ \neg$*signed*$(d, p') \ \vee \ \neg$*trusts*$(p, p'))$

This further yields the following anti-subgoal:

AntiGoal [Sourcing Integrity Breach]
FormalDef $\exists p \in$ *Principal*, $d \in$ *Data* :
sourced$(p, d) \wedge \neg(\forall p' \in$ *Principal* :
$(\forall sig \in$ *Signature* $: \ \neg$*keyBy*$(sig, p) \ \vee \ \neg$*hashBy*$(sig, d)) \vee$
$(\forall ca \in$ *CA* $: \ \neg$*trusts*$(p, ca) \ \vee \ \neg$*trusts*$(ca, p')))$

This is implies that the sourced data either had not been signed by a trusted principal, or it is signed by a principal that cannot be trusted due to the lack of a trust chain.

2.3.5 Single-Sign On Goal Model

Single-Sign On (SSO) is a method of authentication that associates a unique identifier with every user in a VO or across VOs at authentication time such that changes made in VO resource configurations are transparent to the user and such that VO resources have a common identification mechanism for all users.

The requirement to have SSO in project XtreemOS was identified in D3.5.2 [1] as R82. Here, we define the informal SSO goal model as in Figure 5. We first introduce a few useful predicates.

- *User*: The set of grid users. We assume that *User* $\subseteq$ *Principal*.
- *ID*: The set of user ids.

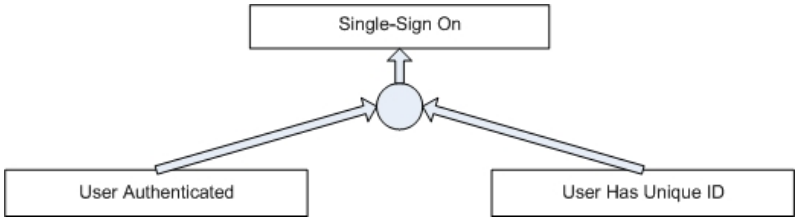

Figure 5: The goal model for SSO.

- *authenticated* : $User \times VO \rightarrow \mathbb{B}$ is a predicate to state whether a user has been authenticated or not by a VO.
- *single_id* : $User \times VO \rightarrow \mathbb{B}$ is a predicate to state whether a user has a single id or not within a VO.

The formal model is then defined as the following goal, which introduces the *sso* : $User \times VO \rightarrow \mathbb{B}$ predicate to denote whether a user enjoys SSO within a VO or not:

Goal [Single-Sign On]
FormalDef $\forall u \in User, vo \in VO$:
$sso(u, vo) \Rightarrow (authenticated(u, vo) \wedge single\_id(u, vo))$

The user authentication and single user id predicates are defined as follows.

Definition [User Authenticated]
FormalDef $\forall u \in User, vo \in VO$: $authenticated(u, vo) \Leftrightarrow (\exists cr \in Credential$:
$binding(u, id, owns, cr) \wedge issued\_By(cr, vo))$

Definition [User Has Unique ID]
FormalDef $\forall u \in User, vo \in VO, cr, cr' \in Credential, id, id' \in ID$:
$single\_id(u, vo) \Leftrightarrow (binding(u, id, owns, cr) \wedge binding(u, id', owns, cr') \wedge$
$issued\_By(cr, vo) \wedge issued\_By(cr', vo) \Rightarrow id = id')$

In the first definition, a user is considered to be authenticated in a VO if and only if there is a credential binding the user to its identity (such as a certificate) and the credential is issued by the VO. The second definition states that a user must not have two different identities within a VO, even though it may have multiple (different) certificates issued by that VO.

2.3.6 Single-Sign On Anti-Goal Model

The informal model of SSO anti-goal is illustrated in Figure 6. As in the case

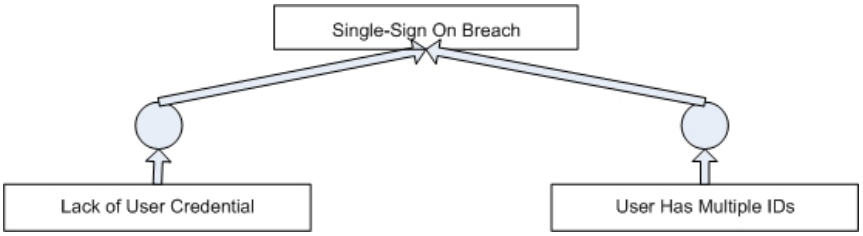

Figure 6: The anti-goal model for SSO.

of confidentiality and integrity, we define the formal anti-goal model of SSO by negating the top-level goal:

AntiGoal [Single-Sign On Breach]
FormalDef $\Diamond\,(\exists u \in \mathit{User}, vo \in \mathit{VO}:$
$\mathit{sso}(u, vo) \wedge (\neg \mathit{authenticated}(u, vo) \vee \neg \mathit{single\_id}(u, vo))$

Expanding the anti-subgoal along the definitions of the authentication and single user id predicates, we obtain the following two vulnerabilities:

AntiGoal [Lack of User Credential]
FormalDef $\Diamond\,(\exists u \in \mathit{User}, vo \in \mathit{VO}:$
$\mathit{sso}(u, vo) \wedge ((\forall cr \in \mathit{Credential}:\ \neg \mathit{binding}(u, id, \mathit{owns}, cr)\ \vee\ \neg \mathit{issued\_By}(cr, vo)) \vee$
$\neg \mathit{single\_id}(u, vo))$

AntiGoal [User has Multiple IDs]
FormalDef $\Diamond\,(\exists u \in \mathit{User}, vo \in \mathit{VO}:$
$\mathit{sso}(u, vo) \wedge (\neg \mathit{authenticated}(u, vo) \vee (\mathit{binding}(u, id, \mathit{owns}, cr) \wedge$
$\mathit{binding}(u, id', \mathit{owns}, cr') \wedge \mathit{issued\_By}(cr, vo) \wedge \mathit{issued\_By}(cr', vo) \wedge \neg(id = id')))$

The former represents the case of a user who is capable of obtaining the SSO capability but that does not have a valid identity credential issued by the VO. The second however refers to the case of the user who has managed to obtain multiple identities but it is still considered to have the SSO capability. This latter case is dangerous as it may lead an SSO-based system to assume that there are two users, and hence leads to a form of non-existent user attack.

2.4 Discussion on the Security Requirements Analysis

The modelling of security requirements using the goal and anti-goal models of the KAOS methodology revealed interesting results as well as enriched our understanding of the security requirements underlying the XtreemOS operating system. In particular, one may summarise the outcome in the following paragraphs:

- The formalisation of the data storage confidentiality and integrity requirements revealed that these requirements focus on one aspect only: access control. In R78 and R80, both confidentiality and integrity are assumed to be achieved through maintaining authorised access to the data. There are other means by which these properties can be maintained, for example, the use of cryptography or data isolation.

- The anti-goal model helped reveal a number of interesting threat and vulnerability scenarios:

    1. Confidentiality: A principal may be able to break the system by gaining false ownership, membership or task assignment despite the lack of valid necessary credentials. Therefore, it is recommended that the system ensures that the credential mechanism (both at producer and at consumer) is robust and cannot be bypassed. Also, it is recommended that the credentials are not issued unnecessarily and are well-defined, since not upholding the latter property of credentials may lead to masquerading attacks.
    2. Integrity: One of the interesting integrity vulnerabilities revealed by the anti-goal model was the possibility that the system may attribute some data to a principal who has not signed it. In other words, the system may misjudge the origin of the data.
    3. SSO: The interesting vulnerabilities that we found in the case of SSO was that the system may issue multiple identities for the same user or the user may lack the necessary identity credential.

3 Security Protocol Verification

Protocol verification refers to the process of analysing, at static time, a protocol in a formal and rigorous manner in order to prove/disprove certain properties about that protocol. The verification of security protocols often aims at establishing that the protocol maintains certain security properties, such as the secrecy of session keys. These properties are usually expressed as safety properties in which the protocol will *always* maintain the property or as *liveness* properties, in which the protocol will eventually arrive at some state satisfying the property.

In the following sections, we shall discuss security properties for the XtreemOS mutual authentication protocol from the above perspective.

3.1 On Security Protocol Verification and its Importance for Distributed and Operating Systems

Security protocols constitute an important and vital element in the infrastructure needed for the secure communication and processing of information in distributed systems. The increasing complexity of these systems and their security requirements creates a need for more robust and secure protocols that ensure their purpose is met with minimum vulnerabilities and threats from malicious attackers. Therefore, it often becomes necessary to treat such protocols as critical components of the system that require rigorous logical methods to verify their security properties.

3.2 A Review of Technologies for Protocol Verification

The area of formal protocol verification comprises a wide range of technologies such as abstract interpretation, theorem proving and model checking that vary according to the degree of automation, the ability to handle complex systems and the expressive power that allows different security properties to be verified.

3.2.1 Abstract Interpretation

In abstract interpretation, a model of the system is defined using a formal language, such as a process algebra, after which the semantics of the language are defined in terms of an abstract semantic domain with a finite size instead of the usual interpretation carried out in a concrete and precise domain but that may have an infinite size. The idea of abstract interpretation was first formalised by Cousot and Cousot in [10] and since then, it has been used widely for the verification of security protocols [6, 8, 13, 16]. In abstract interpretation, the results obtained are sound but approximate in the sense that any vulnerability or attack present in the protocol is necessarily captured by the analysis. However, the analysis may

sometimes produce false positives, i.e. attacks or vulnerabilities that do not exist in the protocol. For a good reference on the principles of abstract interpretation and other static analysis methods, the reader is referred to [17].

One efficient abstract interpretation-based static analysis tool is ProVerif [7] (`http://www.proverif.ens.fr/`). The tool models a security protocol using Horn-clauses and uses abstract interpretation to achieve analyses for unbounded number of sessions for the protocol and it considers a model of the attacker based on Dolev-Yao's most powerful attacker [12]. This attacker is capable of receiving and sending messages over public channels and also applying any cryptographic operations over those messages. ProVerif is also equipped with a translator that allows it to accept as input models written in message-passing process algebra (such as the pi-calculus [15]). This ability renders the tool very attractive since message-passing process algebra are naturally suitable to the formal modelling of security protocols, the latter being based on message-passing. For this reason, and the fact that ProVerif is one of the most efficient and expressive analysis tools specially designed for security protocols, we shall be adopting ProVerif for the analysis to follow.

3.2.2 Model Checking

In model checking, a model of the system is constructed in the input formal language of the model checker after which the state space of the system is explored in search of possible traces than satisfy the property being verified often expressed as a logical formula. In particular, the system corresponds to a *finite state machine*, i.e., a direct graph, in which the nodes represent the states of the system and the edges are transitions between those states. Atomic propositions then are used to represent properties that hold at a certain point of the computation. If the state space is finite, the model checking problem reduces to a graph search. However, in the case of complex systems, the state space tends to be very large or even infinite. In such cases, it is necessary to utilise certain techniques to avoid state space explosions. For example, in *symbolic model checking*, the graph corresponding to the state space is never built but rather encoded as a propositional logical formula. In *partial order reduction*, the number of concurrent process interleaving is reduced by not considering certain unnecessary interleaving. On the other hand, sound but incomplete *abstraction* techniques are used to construct a simplified model of the system.

The area of model checking comprises a wide range of research and tools. We refer the reader to [5, 19], which provide excellent overviews of some of the model checking research and tools specialised in the area of security protocol verification. More recent projects in the area include the AVISPA project (http://www.avispa-project.org/), which is also concerned with the issue of "speed-

ing up the development of the next generation of security protocols and to improve their security". Tools for finding flaws or asserting correctness of a protocol should be completely automated and easily usable, to be integrated in the protocol development and standardization processes. Their main concern is that (semi-)automated protocol analyzers have been proposed in the past recent years, but those perform automatic analysis for small and medium-scale protocols only. The AVISPA Library comprises more than one hundred security problems derived several protocols.

3.2.3 Automated Theorem Proving

In automated theorem proving, a computer program is used to show that some statement called the conjecture is a logical consequence of a set of other statements called axioms and hypotheses. The language of conjectures, axioms and hypotheses is usually logic-based, such as first order logic or some higher order logics. In the problem of automated proof verification is concerned certifying that the proof of a theorem is a valid one. This then requires that each step of the proof be verified by a program or a primitive recursive function, and hence the problem should always be decidable. However, in reality, this is not always possible due to *Gödel's Incompleteness Theorem*. The implication of this theorem is that there are logical questions that require an unbounded amount of resources to solve them. Therefore, theorem provers may fail to terminate and are usually designed to be interactive programs that require the intervention of the user.

Theorem proving techniques have been applied to the problem of security property verification in cryptographic protocols. A survey of these techniques and associated logics can be found in [5].

3.2.4 Technology Review Conclusion

Our choice of approach was mainly driven by the following main considerations:

- The natural suitability of the input language of the tool for modelling security protocols. Such a language has to be message-passing and capable of expressing security primitives (such as cryptographic functions).

- The presence of supporting tools and the degree of automation of those tools as well as their efficiency and natural ability in verifying security properties.

- The approach as well as the supporting tools must be able to deal with infinite runs of the system, since security protocols are assumed to have an infinite number of sessions.

After considering the different approaches (i.e. abstract interpretation, model checking and theorem proving) in the area of static analysis of security protocols, we found that the abstract interpretation using process algebra such as the π-calculus was very well suited to the modelling of security protocols. Furthermore, this approach is supported by one of the most efficient static analysis tools targeted at the analysis of security protocols, namely ProVerif.

3.3 The XtreemOS Mutual Authentication Protocol

This section presents a revised version of the mutual authentication protocol described in the first specification of security services (refer to D3.5.3 [2]). The protocol is based on the classic Diffie-Hellman key agreement protocol[11]. At the end of this protocol, a shared secret key is agreed between communicating parties who (a) previously are unknown to each other; and (b) are under two different administrative domains, who may or may not use the same kind of authentication methods. This protocol aims to prove a user's identity in the context of a VO.

The following is a list of notations used in the protocol:

- U: a user within a Home Authentication Authority (HAA)
- $User_{id}$: the user's unique identity within the HAA
- VO_{id}: the identity of a VO that U is registered with
- N: a resource node[1]
- VOM: a VO management authority that runs CDA and X-VOMS
- g, n: the Diffie-Hellman (DH) parameters[2]
- R_Y: a random number generated by entity Y
- G_Y: a constant where $G_Y = g^{R_Y} \bmod n$
- G_{YX}: a constant where $G_{YX} = (g^{R_Y} \bmod n)^{R_X} \bmod n$
- T_Y: a timestamp generated by entity Y
- K_Y^r: the private key of entity Y
- K_Y^u: the public key of entity Y

[1] Examples of a node are a resource node for application execution or a MRC or OSD node in XtreemFS. In fact, any service that has a public key certificate can be such a node.

[2] g is the DH exponent (i.e. generator) and n is the size of the DH field which the computation is based upon.

- <M>K^r_Y: a message M signed by Y's private key
- $\{M\}K^u_Y$: a message M encrypted by Y's public key
- $RndMsg_Y$: a random message generated by entity Y

The mutual authentication protocol consists of the following messages described in the classical Alice-Bob style:

1. $U \rightarrow VOM$: $User_{id}$, VO_{id}, g, n, G_U
2. $VOM \rightarrow U$: <$User_{id}$, VO_{id}, g, n, G_U, T_{VOM}> K^r_{VOM}
3. $U \rightarrow N$: { <$User_{id}$, VO_{id}, g, n, G_U, T_{VOM}>K^r_{VOM}, $RndMsg_U$, T_U }K^u_N
4. $N \rightarrow U$: G_N, $\{RndMsg_U, T_U\}G_{UN}$,$\{RndMsg_N, T_N\}G_{UN}$
5. $U \rightarrow N$: $\{RndMsg_N, T_N\}G_{NU}$

The protocol commences when the user, U, in (1.) requests from its home VO management authority, VOM, an XOS certificate by submitting to VOM its user identity U_{id}, the id of the VO it belongs to, V_{id}, the Diffie-Hellman parameters, g, n, and the constant G_U, which the user computes based on a fresh random number, R_U, it generated. This initial message is assumed to be communicated over a trusted secure channel shared between U and VOM. In general, there are three ways in which such a trusted and secure channel can be established (refer to D3.5.3, Section 4.2.4). As long as the secrecy and authenticity requirements are met, this step is not constrained to any specific authentication methods. Once VOM receives the message, it will check the authenticity of the user and the validity of its VO membership.

In the next message (2.), VOM replies to U by sending it an XOS certificate signed by its private key, K^r_{VOM}, and carrying a timestamp, T_{VOM}, denoting the expiry time of the certificate. U is then able to use the certificate within its time validity to authenticate itself to any node in the VO that trusts VOM and to establish a shared session with that node.

In message (3.), U contacts one such node, N, over an insecure public channel. U sends to N the certificate it received from VOM along with a timestamped message, $RndMsg_U, T_U$, all encrypted with the public key of the node, K^u_N. This ensures that the message is fresh and that it can only be decrypted by N. Once N receives the message, it checks the validity of the certificate and if it is valid[3], it then generates the public constant, G_N, as well as the session key G_{UN}.

[3] Validity needs to be formally defined

In message (4.), N then sends to U the public constant G_N along with the original timestamped message of U encrypted under the session key G_{UN} and a new timestamped message, $RndMsg_N, T_N$ generated by N and encrypted with G_{UN}. Upon the receipt of this message, U generates its own copy of the session key, G_{NU}, which is equivalent to G_{UN}. U then uses G_{NU} to decrypt the two parts of the message containing $RndMsg_U, T_U$ and $RndMsg_N, T_N$. If $RndMsg_U, T_U$ is the same as the original and $RndMsg_N$ is fresh (i.e. T_N is recent), then N is authenticated and G_{NU} is accepted as the session key by U.

Finally, in message (5.), U sends to N message, $RndMsg_N, T_N$, encrypted with its session key G_{NU}. N then receives the message, decrypts it, and then checks that the pair $RndMsg_N, T_N$ is the same as the original one generated by N. If this is the case, then N accepts the authenticity of U and the use of G_{UN} as the session key. At the end of this step, both U and N will have authenticated themselves and accepted $G_{NU} = G_{UN}$ as their session key, only known by them.

3.4 Formal Model and Analysis of the Protocol

In this section, we formally model and analyse the mutual authentication protocol introduced in the previous section. Our approach is to first define what we mean by mutual authentication. Then we construct a model of the protocol in a formal language that is expressive enough to be able to capture concepts and mechanisms used in the protocol and that is supported by automated verification tools. Finally, we use the verification tool(s) to verify that the mutual authentication property is upheld by the protocol and that external attackers are unable to break the property.

3.4.1 A Formal Definition of Mutual Authentication

Our definition of authentication is based on the type (3) authentication as specified by Lowe [18, §3.3]. In this type, two entities, a and b can mutually authenticate themselves and agree on additional information specific to the protocol session if each is convinced that the other entity has participated in the run and that the information exchanged is the same. This is achieved using the concept of *commit* and *running* events. The commit-running events provide a mechanism to ensure the temporal ordering of protocol steps in a manner leading to (mutual) authentication. The occurrence of a commit event in b must imply that the running event in a has already occurred. This means that b has authenticated a. The opposite authenticates b to a. The commit event must happen after the end of the protocol, whereas the running event can happen at anytime during the protocol but must happen before it is over.

More formally, it is possible to define one-way authentication as follows.

Definition 1 (One-Way Authentication) *Given two processes, A and B, we say that A* authenticates *B* agreeing on *M if A executes event $commit_A(M)$ and B executes event $running_B(M')$ and the following is true:*

$$(commit_A(M) \Rightarrow running_B(M')) \ \wedge \ (M = M')$$

In other words, the definition of one-way authentication states that the occurrence of $running_B(M')$ precedes the occurrence of $commit_A(M)$. Furthermore, by the time both these events have occurred, their corresponding parameters, M, M', must be the same according to some definition of the $=$ relation.

Now, the definition of mutual authentication is formalised as follows, based on the definition of one-way authentication.

Definition 2 (Mutual Authentication) *We say that A and B* mutually authenticate each other*, if the following holds true:*

$$A \text{ authenticates } B \text{ agreeing on } M \ \Leftrightarrow \ B \text{ authenticates } A \text{ agreeing on } M'$$

Here, it is not necessary that $M = M'$, however, in session key agreement protocols where the aim is to establish a common session key, the two values must agree if they represent the common session key. In fact, in our analysis to follow, this will be the case.

3.4.2 The Language

The syntax of the input language is given in Figure 7. This syntax is based largely on a version of the π-calculus [15] called the applied π-calculus [4], which extends the π-calculus with functional constructors/destructors and equational theories defining how constructors and destructors are related to each other. The meaning of the syntax is described informally as follows: A term, ⟨term⟩, is either an identifier, a sequence of terms or the application of a function to a sequence of terms. A fact, ⟨fact⟩, is either a predicate applied to a sequence of terms, an inequality check of two terms or an equality check of two terms. Using terms and facts, a process is then defined according to the following constructs:

- `(` ⟨process⟩ `)` : a process enclosed by two brackets to remove ambiguity.
- `!` ⟨process⟩: a replicated process, which is capable of spawning as many copies of itself as is required by the context.
- `0`: the null process, which is incapable of any behaviour and cannot evolve any further.

⟨term⟩	::=	⟨ident⟩
	|	(seq⟨term⟩)
	|	⟨ident⟩ (seq⟨term⟩)
⟨fact⟩	::=	⟨ident⟩ : seq⟨term⟩
	|	⟨term⟩ <> ⟨term⟩
	|	⟨term⟩ = ⟨term⟩
⟨process⟩	::=	(⟨process⟩)
	|	! ⟨process⟩
	|	0
	|	new ⟨ident⟩; ⟨process⟩
	|	if ⟨fact⟩ then ⟨process⟩ [else ⟨process⟩]
	|	in(⟨term⟩, ⟨term⟩)[; ⟨process⟩]
	|	out(⟨term⟩, ⟨term⟩)[; ⟨process⟩]
	|	let ⟨term⟩ = ⟨term⟩ in ⟨process⟩ [else ⟨process⟩]
	|	⟨process⟩ | ⟨process⟩
	|	event ⟨term⟩[; ⟨process⟩]

Figure 7: The syntax of processes.

- new ⟨ident⟩; ⟨process⟩: the process that creates a new identifier with scope restricted to the residual process.

- if ⟨fact⟩ then ⟨process⟩ [else ⟨process⟩]: a process that checks whether the specified fact is true. If so, it chooses the then-branch. Otherwise, it proceeds as the else-branch.

- in(⟨term⟩, ⟨term⟩)[; ⟨process⟩]: an input process that receives a term over a channel name (the first indicated term) and uses that term to replace its input parameter (the second indicated term). It then proceeds as the residual process. The input parameter has a scope restricted to the residual process.

- out(⟨term⟩, ⟨term⟩)[; ⟨process⟩]: an output process, which sends over a channel (the first indicated term) a message (the second indicated term) and then proceeds as the residual process.

- let ⟨term⟩ = ⟨term⟩ in ⟨process⟩ [else ⟨process⟩]: a let-process, which assigns a term (the second indicated) to another (the first indicated) with the scope of the residual process. If the assignment fails, the let-process proceeds as the else-process indicated at the end.

- ⟨process⟩ | ⟨process⟩: the parallel composition of two processes, which has an interleaving semantics.

- `event` ⟨term⟩[`;` ⟨process⟩]: an event that has a name and a possible sequence of terms that it may synchronise on. The event is followed by the residual process.

For a formal definition of the semantics of the syntax, we refer the reader to [7] and [4].

3.4.3 The Model

We define here a model of the mutual authentication protocol as described in Section 3.3 using the syntax of the previous section. The protocol model is shown in Figure 8.

```
(* The VOM Process Definition *)
let vom =
(* Get initialised with the VOM private key *)
    in(tvom, skVOM0);
(* Receive a Xcert request from a user *)
    in(vom, (v1, v2, v3, v4, v5));
(* Check that the user's name is u - (identification abstraction) *)
    if v1=u then
(* Send the user's Xcert signed by VOM's private key and timestamped *)
    out(vom, sign((v1, v2, v3, v4, v5, Tvom), skVOM0))


(* The User Process Definition *)
let p0 =
(* Create a new user random number *)
    new Ru;
(* Request an XCert from the VOM *)
    out(vom, (u, vid, DHexp, DHfld, g(Ru))); in(vom, xcert);
    !(
(* Get initialised with a node's name and public key *)
    in(tu, (nv, pkN0));
(* Create a new message *)
    new MSGu;
(* Contact the node and wait for the response *)
    out(nv, pubenc((xcert, MSGu, Tu), pkN0)); in(u, (m1, m2, m3));
(* Generate the session key *)
    let ku = f(m1, Ru) in
(* Decrypt the node messages using the session key *)
    let m4=dec(m2, ku) in
    let m5=dec(m3, ku) in
(* Check that the message returned by the node is the same one sent by the user *)
(* If so, signal a running event *)
    if m4 = (MSGu, Tu) then event p0running(f(g(Rn), Ru));
(* Send to the node its own message back and
signal a commit event agreeing on the session key *)
    out(nv, m3);event p0commit(f(g(Rn), Ru)))
```

Figure 8: The model of the mutual authentication protocol.

```
(* The Node Process Definition *)
let p1 =
    !(
   (* Initialise the node with its private key and the VOM public key *)
      in(tn, (skN1, pkVOM1));
   (* Receive a request from the user *)
      in(nv, x);
   (* Decrypt the message sent by the user using the node's private key *)
      let (x0, x1, x11) = pubdec(x, skN1) in
   (* Verify the XCert signed by VOM using the latter's public key *)
      let (x2, x3, x4, x5, x55, x555) = checksign(x0, pkVOM1) in
   (* Check that the VO id is correct and that the timestamp is fresh *)
      if x3 = vid then if x4 = DHexp then
      if x5 = DHfld then if x555 = Tvom then
   (* Create a new random number and message *)
      new Rn; new MSGn;
   (* Create the session key signal the running event agreeing on the session key *)
      let kn = f(x55, Rn) in event p1running(f(g(Ru), Rn));
   (* Send message to the user including g(Rn) *)
      out(x2, (g(Rn), enc((x1, x11), kn), enc((MSGn, Tn), kn)));
   (* Receive the reply from the user *)
      in (nv, x6);
   (* Decrypt the message *)
      let x7 = dec(x6, kn) in
   (* Check that the received message is the same as the original one sent *)
   (* If so, signal the commit event agreeing on the session key *)
      if x7 = MSGn then event p1commit(f(g(Ru), Rn)))


(* The Protocol Definition *)
   (* Create secure private channel names *)
new tvom; new vom; new tn; new tu; (
   (* Create private/public key pair for VOM *)
new skVOM; let pkVOM = pk(skVOM) in
   (* Initialise VOM with its private key and advertise its public key *)
out(tvom, skVOM); out(attc, pkVOM);
   (* Initialise as many nodes with their private key and VOM's public key *)
   (* Advertise the node's public key and initialise the user with the node's public key *)
!(
new skN;
let pkN = pk(skN) in out(attc, pkN);
(out(tn, (skN, pkVOM)) | out(tu, (n, pkN)))
 ))
   (* Run the node, the user and the VOM process concurrently *)
   (* The attacker is listening on channel attc *)
| (p1) | (p0) | (vom) | (!(in(attc, v)))
```

Figure 5: The model of the mutual authentication protocol (Cont.).

The protocol definition consists of three process definitions: *vom* representing the VO management process, $p0$ representing the user process and $p1$ representing the node process. The protocol definition starts creating the private and public parts of the *vom* process and initialises it with the private part of the key. It also advertises the public part over a public channel, *attc*, that can be read by the attacker. It then starts initialising any number of nodes with their private keys and with the public key of *vom* as well as advertising the public key of these nodes over *attc*. At the same time, the protocol process runs the node, the user and vom processes in parallel with each other.

The *vom* process is ready to accept a request from a user for an XOS certificate over a channel, *vom*, known only to the vom and user processes. The request is dealt with by sending to the user a signed certificate and then signaling using an event the termination of vom. The assumption here is that there is a single user willing to communicate with several nodes. The user then, after receiving the certificate, starts a replicated process which is able to start mutually authenticating as many nodes as available. Note the presence of the commit-running events for both the user and node processes, which will be used to verify the success of authentication.

The specification of the protocol utilises the following equations defining the relationship among functions:

$$\mathit{getmess}(\mathit{sign}(m, k)) = m \tag{1}$$

$$\mathit{checksign}(\mathit{sign}(m, k), pk(k)) = m \tag{2}$$

$$\mathit{pubdec}(\mathit{pubenc}(x, pk(y)), y) = x \tag{3}$$

$$\mathit{dec}(\mathit{enc}(x, y), y) = x \tag{4}$$

$$f(g(y), x) = f(g(x), y) \tag{5}$$

Where equations 1-3 define public-key cryptography operations, equation 4 defines secret-key cryptography and equation 5 defines the Diffie-Hellman pair of functions. This representation of the Diffie-Hellman functions in this manner is popular in process-algebraic-based protocol definitions [3, 7] and it's motivated by the analysis algorithms. Essentially, $g(R_Y) = G_Y$ and $f(G_Y, R_X) = G_{YX}$ in terms of the G_Y and G_{YX} defined in Section 3.3. The user process, however, also passes to the vom process the Diffie-Hellman parameters, $DHexp$ and $DHfld$, as numbers representing the exponent and the field size. This is included to model faithfully the protocol description.

3.4.4 The Data Leakage Analysis

The first analysis of the XtreemOS mutual authentication protocol using ProVerif was carried out to determine which data terms can possibly be leaked to the exter-

nal Dolev-Yao attacker. The results of the leakage analysis are given in Table 6.

TERM	LEAKAGE STATUS
Ru - random number	Not leaked
Rn - random number	Not leaked
$f(g(Ru), Rn)$ - session key	Not leaked
$f(g(Rn), Ru)$ - session key	Not leaked
$g(Ru)$ - data	Not leaked
$MSGu$ - message	Not leaked
$MSGn$ - message	Not leaked
skN - private key	Not leaked
$skVOM$ - private key	Not leaked
Tu - timestamp	Not leaked
Tn - timestamp	Not leaked
$Tvom$ - timestamp	Not leaked
tu - secure channel	Not leaked
tn - secure channel	Not leaked
vom - secure channel	Not leaked
$tvom$ - secure channel	Not leaked
pkN - public key	Leaked
$pkVOM$ - public key	Leaked
$DHexp$ - Diffie-Hellman data	Leaked
$DHfld$ - Diffie-Hellman data	Leaked
u - public channel	Leaked
vid - public channel	Leaked
nv - public channel	Leaked
$attc$ - public channel	Leaked
$g(Rn)$ - data	Leaked

Figure 6: Leakage analysis for the XtreemOS mutual authentication protocol.

From these results, it is clear that the attacker was not able to construct or capture the session key, $f(g(Ru), Rn)$, which is the same as $f(g(Rn), Ru)$ nor indeed any of the terms that need to be kept secret such as the random numbers generated by the user and the node, R_u, R_n, the private keys of the node and VOM, skN, $skVOM$, and the exchanged messages generated by the user and the node, $MSGu$, $MSGn$.

On the other hand, the analysis reveals that the attacker is able to capture the term, $g(Rn)$, which is a representation of G_N in the protocol's Alice-Bob

description of Section 3.3. This implies that the attacker may be able to guess the value of the random number Rn if the node did not choose a sufficiently large Rn (in the order of 100 digits long). Similarly, the attacker may also guess the value of the session key $f(g(Rn), Ru)$ given Rn, if the user did not manage to choose a sufficiently large random number Ru (again in the order of 100 digits long).

On the other hand, we note that the attacker is unable to capture $g(Ru)$ generated by the user, even though this term is sent on the clear initially to the VOM. This is due to the fact that the *vom* channel is declared as a secure channel using the restriction (`new` *vom*) in the definition of the protocol. Even though in the model, such channels are possible to describe, in reality, one cannot assume that there are channels secure by their nature. Therefore, our model is somehow unrealistic in that it assumes the existence of such naturally secure channels. One alternative to move away from this assumption would be to use encryption for initial communication between the use and VOM. In that case, it would be possible to send the message (including $g(Ru)$) over an insecure public channel.

3.4.5 The Mutual Authentication Analysis

For the second, we wanted to verify the mutual authentication property for the protocol using commit-running events. The ProVerif tool was able to prove that the following two implications are true:

$$\textit{p1commit}(f(x55, Rn)) \Rightarrow \textit{p0running}(f(m1, Ru)) \quad (6)$$

$$\textit{p0commit}(f(m1, Ru)) \Rightarrow \textit{p1running}(f(x55, Rn)) \quad (7)$$

Where $x55$ is a term instantiated to $g(Ru)$ and $m1$ is a term instantiated to $g(Rn)$. According to the equation $f(g(x), y) = f(g(y), x)$, we can infer that $f(x55, Rn) = f(m1, Ru)$ in both cases, therefore we can say that the above result satisfies the definition of mutual authentication (Section 3.4.1) for both the node and the user.

3.5 Discussion on Protocol Verification in XtreemOS

We found that the modelling of the XtreemOS mutual authentication protocol was naturally straightforward in the message-passing process algebra due to the similarity in the nature of the protocol and the process algebraic language. The existence of an automatic and efficient verification tool backed up by a formal theory (in the style of Horn clauses [14]) facilitated the task of verifying security properties such as data leakage and mutual authentication related to our protocol.

Both the data leakage and the mutual authentication analyses revealed expected results. One recommendation is related to the fact that some of the data terms, such as $g(Rn)$, are sent on the clear and captured by the attacker. This requires that the user and the node must choose large random numbers, Rn, Ru, in

order to prevent the attacker from guessing the session key. This is a well-known requirement in Diffie-Hellman-based protocols that reveal some of the protocol-generated data on the clear.

Another recommendation is related to the assumption of secure communication channels, for example in the initial communication between the user and VOM. Although this assumption may seem realistic for the specific case where both the user and VOM are running within the same trust domain, it is unrealistic for the general case where there is an untrusted network separating the user and VOM. This aspect of the protocol was clarified when modelling the protocol in the language of applied π-calculus, where it was necessary to use the restriction operator *new* for hiding channel *vom* in order to express the security property of the channel without using cryptographic operations.

4 Conclusions

This document describes the formal analysis of two elements in the development process of XtreemOS security system: security requirements and protocol verification.

In relation to requirements, we have modelled confidentiality, integrity, and single sign-on properties following the KAOS goal-oriented requirements-engineering methodology. The following recommendations summarise the outcome of this exercise:

- The formalisation of the data storage confidentiality and integrity requirements revealed that these requirements focused only on one aspect of confidentiality: access control. There are other means by which these properties can be maintained, for example, the use of cryptography or data isolation.

- The application of anti-goal modelling revealed some potential threat and vulnerability scenarios. A principal may be able to break the system by gaining false ownership, membership or task assignment despite the lack of valid necessary credentials. Therefore, it is recommended that the system ensures that the credential mechanism (both at producer and at consumer) is robust and cannot be bypassed. Other scenarios revealed the possibility of masquerading attacks and mis-judgment in the origin of the data.

In relation to protocol verification, we have modelled the XtreemOS mutual-authentication protocol using the Applied π-calculus, and verified automatically using the ProVerif tool that the protocol holds the mutual-authentication property. The following recommendations summarise the outcome of this exercise:

- Some of the data terms, such as $g(Rn)$, are sent on the clear and captured by the attacker. This requires that the user and the node must choose large random numbers, Rn, Ru, in order to prevent the attacker from guessing the session key. This is a well-known requirement in Diffie-Hellman-based protocols that reveal some of the protocol generated data on the clear.

- Another recommendation is related to the assumption of secure communication channels. Although this assumption may seem realistic for the specific case where both the user and VOM are running within the same trust domain, it is unrealistic for the general case where there is an untrusted network separating the user and VOM.

As future work, we are planning to generate automatically test-cases from the requirements model.

References

[1] XtreemOS Deliverable D3.5.2: Security Requirements for a Grid-based OS, 2007.

[2] XtreemOS Deliverable D3.5.3: First Specification of Security Services, 2007.

[3] Martín Abadi, Bruno Blanchet, and Cédric Fournet. Just fast keying in the pi calculus. In David Schmidt, editor, *Programming Languages and Systems: Proceedings of the* 13^{th} *European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 340–354, Barcelona, Spain, March 2004. Springer Verlag.

[4] Martín Abadi and Cédric Fournet. Mobile Values, New Names, and Secure Communication. In *Proceedings of the* 28^{th} *ACM Symposium on Principles of Programming Languages*, pages 104–115, London, UK, January 2001. ACM Press.

[5] Benjamin Aziz, David Gray, Geoff Hamilton, Frederic Oehl, James Power, and David Sinclair. Implementing Protocol Verification for E-Commerce. In *Proceedings of the* 2^{nd} *International Conference on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet*, L'Aquila, Italy, August 2001.

[6] Benjamin Aziz, Geoff Hamilton, and David Gray. A static analysis of cryptographic processes: The denotational approach. *Journal of Logic and Algebraic Programming*, 64(2):285–320, August 2005.

[7] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the* 14^{th} *IEEE Computer Security Foundations Workshop*, pages 82–96, Cape Breton, Nova Scotia, Canada, July 2001.

[8] Chiara Bodei, Pierpaolo Dagano, Flemming Nielson, and Hanne Riis Nielson. Static analysis for the π-calculus with applications to security. *Information and Computation*, 168(1):68–92, July 2001.

[9] Common Criteria Consortium. Common criteria for information technology security evaluation, September 2006.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the* 4^{th} *ACM Symposium on Principles of*

Programming Languages, pages 238–252, Los Angeles, California, U.S.A., January 1977. ACM Press.

[11] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[12] Danny Dolev and A. Yao. On the security of public key protocols. In *Proceedings of the* 22^{nd} *Annual Symposium on Foundations of Computer Science*, pages 350–357, October 1981.

[13] Jérôme Feret. Confidentiality analysis of mobile systems. In *Proceedings of the* 7^{th} *International Static Analysis Symposium*, volume 1824 of *Lecture Notes in Computer Science*, pages 135–154, University of California, Santa Barbara, USA, June 2000. Springer Verlag.

[14] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, March 1951.

[15] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes (parts I & II). *Information and Computation*, 100(1):1–77, September 1992.

[16] Flemming Nielson, René Rydhof Hansen, and Hanne Riis Nielson. Abstract interpretation of mobile ambients. *Science of Computer Programming*, 47(2–3):145–175, May 2003.

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

[18] Peter Ryan, Steve Schneider, Michael Goldsmith, Gavin Lowe, and Bill Roscoe. *Modelling and Analysis of Security Protocols*. Addison-Wesley Professional, December 2000.

[19] P.Y.A. Ryan and S.A. Schnieder. *Modelling and Analysis of Security Protocols*. Addison-Weslley, 2001.

[20] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In *International Conference on Software Engineering*, pages 5–19, 2000.

[21] A. van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. *26th ACM-IEEE International Conference on Software Engineering (ICSE'04)*, pages 148–157, 2004.