Modeling Security Requirements Through Ownership, Permission and Delegation

Paolo Giorgini
University of Trento
giorgini@dit.unitn.it

Fabio Massacci
University of Trento
massacci@dit.unitn.it

John Mylopoulos
University of Toronto
jm@cs.toronto.edu

Nicola Zannone
University of Trento
zannone@dit.unitn.it

Abstract

Security Requirements Engineering is emerging as a branch of Software Engineering, spurred by the realization that security must be dealt with early on during the requirements phase. Methodologies in this field are challenging, as they must take into account subtle notions such as trust (or lack thereof), delegation, and permission; they must also model entire organizations and not only systems-to-be.

In our previous work we introduced Secure Tropos, a formal framework for modeling and analyzing security requirements. Secure Tropos is founded on three main notions: ownership, trust, and delegation. In this paper we refine Secure Tropos introducing the notions of at-least delegation and trust of execution; also, at-most delegation and trust of permission. We also propose monitoring as a security design pattern intended to overcome the problem of lack of trust between actors. The paper presents a semantics for these notions, and describes an implemented formal reasoning tool based on Datalog.

1 Introduction

Recent years have seen many proposals that incorporate security in the software engineering process. At one end of the spectrum, such proposals ensure good coding practices [27]. At the other extreme, the emphasis is on securing the organization within which a software system functions [2]. In either case, modeling and analysis of security requirements has become a key challenge for Software Engineering [6, 8], and is the subject of this paper.

Proposals for Security Requirements Engineering can be classified under one of two classes. Approaches such as [3, 19, 25] use an off-the-shelve modeling and analysis framework – such as UML, KAOS, or i*/Tropos – and model in that framework security requirements. For such approaches, the features of the framework are used to formally analyze security requirements and guide the implementation. Other approaches [9, 16, 20, 22, 24, 26] adopt a requirements engineering framework and enhance it with novel constructs specific to security. For such approaches, formal analysis techniques and implementation guidelines need to be revised and/or extended to accommodate the new concepts.

Most proposals in the literature focus on protection aspects of security and explicitly deal with a series of security services (integrity, availability etc.) and related protection mechanisms (such as passwords, or cryptographic techniques). A shift from this perspective towards early requirements was proposed by the authors [13, 14] who extended the i*/Tropos modeling framework [5] to define Secure Tropos. The proposal introduces concepts such as ownership, trust, and delegation within a requirements modeling framework and shows how security and trust requirements can be derived and analyzed.

The baseline for the contributions of this paper is this work. After a large case study on the compliance of an ISO-17799-like security policy [21] with Italian privacy legislation, it was concluded that the concepts offered by Secure Tropos are the right ones but are too coarse-grained to capture important security facets. Specifically, unlike what the framework provides for, we found that for pragmatic reasons, it is often the case that services and permissions are delegated to actors who are not trusted. Nevertheless, the overall system is still considered secure if there is a way to hold such delegations accountable by monitoring their (wrong) doings.

The second observation is that trust in actors (or lack thereof) comes in different flavors: we may trust an actor to actually deliver the services we require (taking into account skills and/or commitment), or to honor granted permissions. In trust management and authorization settings (e.g. [4, 7, 17]) one only finds delegations of permission (through authorization). Requirements of availability are equally important, however, and can only be captured by modeling delegation of execution (where one actor delegates to another the responsibility to execute a service).

Thus, the key contribution of this paper is a refined framework for modeling and analyzing security require-

ments over what has been proposed in [13, 14]. The refinement includes a distinction of the notions of delegation of execution (at-least delegation) and delegation of permission (at-most delegation), also the distinction of the notions of trust of execution (at-least trust) and trust of permission (at-most trust). Finally, we propose the use of monitoring as a security pattern, a design solution intended to overcome the problem of lack of trust between actors. These constructs have been formalized and can be formally analyzed through a tool-supported process. As done in other frameworks that deal with trust and security [7, 18, 23] we use Datalog as the underlying semantic framework. Other approaches such as [15] propose to use deontic logic to model the notions of obligation, empowering and permission. However, these approaches do not distinguish between direct and indirect capabilities and responsibilities. This is a fundamental distinction in modeling organizations. In fact, one actor could delegate the execution of (permission on) a service to another actor since he is not directly able to do so alone.

In the remainder of the paper, we introduce a running example (§2), discuss the overall framework (§3, §4, and §5), its formal semantics (§6) and some useful formal properties exploited for verification purposes (§7). We conclude with a brief discussion of related work and a summary of our contributions (§8).

2 A Running Example

The example is abstracted from a substantial case study on the compliance of Italian public administrations such as universities, local governments and health care authorities to Italian security and privacy legislation.

In summary, the law requires administrations to set up sophisticated security and privacy policies that are actually quite close to the complexity of the ISO-17799 standard for security management. Dealing with privacy introduces additional complications such as data ownership, trust and consent. Details on the case study for an university can be found in [21].

For readability we introduce here dramatis personae:[1]

Alice is an administrative officer, for example of the teaching evaluation office;
Bob, Bert, and Bill are students;
Sam is (the manager of) the student IT system;
Paul and Peter are professors.

3 Tropos and Secure Tropos

Tropos [5] is a methodology for developing agent-oriented software. The methodology supports different development phases from early requirements to detailed design. The methodology is founded on models that use the concepts of actor, goal, task, resource and social dependency for defining the obligations of actors (dependees) to other actors (dependers). A goal represents the strategic interests of an actor. A task specifies a particular course of actions that produces a desired effect, and can be executed in order to satisfy a goal. A resource represents a physical or an informational entity. Finally, a dependency between two actors indicates that one actor depends on another to accomplish a goal, execute a task, or deliver a resource. Tropos is well suited for modeling both an organization and IT systems operating within it. However, in [12] we have argued that the Tropos framework lacks the ability to capture important aspects of security, and hence the new proposal.

Secure Tropos has been proposed in [13, 14] as a formal extension of Tropos, intended for modeling and analysis of functional and security requirements. To simplify terminology, the notion of *service* is used in this framework to refer to a goal, task, or resource, and three new relationships are introduced:

- *Ownership* (between an actor and a service) represents the fact that an actor is the legitimate owner of a service;
- *Trust* (among two actors and a service), marks a social relationship that indicates the belief of one actor that another actor will not misuse the service he has been granted.
- *Delegation* (among two actors and a service), marks a formal passage of permission.

Example 1 *By law, Bob is the owner of his personal data. Yet, the data is stored on servers that are managed by Sam, who in turn gives access to Alice and Paul. In this scenario, Sam should seek the consent of (or, permission from) Bob for data processing concerning his personal data.*

Another feature of our proposal is the distinction between permission and delegation.

Example 2 *The letter of the University rector that assigns to the CIO the responsibility for enacting privacy protection measures is an example of delegation.*

In digital trust management systems, this would be matched by the issuance of a delegation certificate. The basic consequence of delegation is having more permission holders. In contrast to these notions, trust simply marks a social relationship that is not formalized by a "contract" (such as digital credential or a letter). There might be cases where we might be happy with a "social" protection mechanisms (e.g. because it is impractical or too costly to do otherwise). In other cases, however, formal delegation is essential.

[1] This impersonation is actually closer to reality than one may think: the law requires the assignment of responsibility of each IT sub-system to a person.

4 Refining Delegation and Trust

Now we introduce a conceptual refinement of delegation and trust relationships, that will allow us to capture and model important security facets

Example 3 *Alice is interested in gathering data on student performance, for which she depends on Sam. Bob owns his personal data, such as his academic record. Bob delegates permission to provide information about his academic record to Sam, on condition that his privacy is protected (i.e., his identity is not revealed).*

In this scenario (Fig. 1(a)), there is a difference in the relationships between Alice–Sam and Bob–Sam. This is due to a difference in the type of delegation.

Example 4 *Bob* delegates permission *to Sam to provide only the relevant information and nothing else. On the other hand, Alice, who wants student data,* delegates *the* execution *of her goal to Sam. According to Alice, Sam should at least fulfill the goal she wants. She is not interested in whether Bob trusts Sam, she simply wants information. Bob, on the other hand, worries about authorization: anyone who uses his personal must be authorized to do so.*

If we want to check that requirements are consistent and that security requirements for each actor are met, it is essential to distinguish between these two notions of delegation. We use **at-most delegation** when the delegator wants the delegatee to at most fulfill a service. This is *delegation of permission*, where the delegatee thinks "I have the permission to fulfill the service (but I do not need to)", whereas **at-least delegation** means that the delegater wants the delegatee to at least perform the service. This is the *delegation of execution*. The delegatee thinks, "Now, I have to get this service fulfilled (...let's get started)". In the graphical representation of Fig. 1 we represent these relationship as edges respectively labeled **P** and **E**.

Further, we want to separate the concepts of trust and delegation, as we might need to model systems where some actors must delegate permission or execution to other actors they don't trust. Also in this case it is convenient to have a suitable distinction for trust in managing permission and trust in managing execution. The meaning of **at-most trust** is that an actor (truster) trusts that another actor (trustee) will at most fulfill a service, but will not overstep it. The meaning of **at-least trust** is that an actor (truster) trusts that another actor (trustee) will at least fulfill a service.

Example 5 *At-most trust applies to permissions: Bob trusts Sam to use Bob's personal information within certain bounds. At-least trust applies to executions: Alice believes Sam can accomplish her desired task (and possibly more).*

Actor dependencies in Tropos (and i*) represent at-least delegation combined implicitly with at-least trust. The delegation proposed by Giorgini et al. [14] blurs the distinction between at-least and at-most delegation and at-least and at-most trust.

In the development of a system, a designer should be able to guarantee and implement the trust and delegation relationships captured in the social setting during the requirements analysis phase. This analysis can be done with the methodology advocated in [14].

However, specific situations may impose that some services have to be delegated to some actors even when there is no trust relationship between them. In our example [21], this situation may arise when services are outsourced to outside providers for whom a trust relationship remains to be developed. These providers may offer services that range from cleaning to security, from ERP clock cycles to network backbones. We propose to legitimize such situations in our framework by adopt monitoring as a design pattern that can neutralize the lack of trust. The following section focuses on monitoring as a design pattern.

5 Monitoring

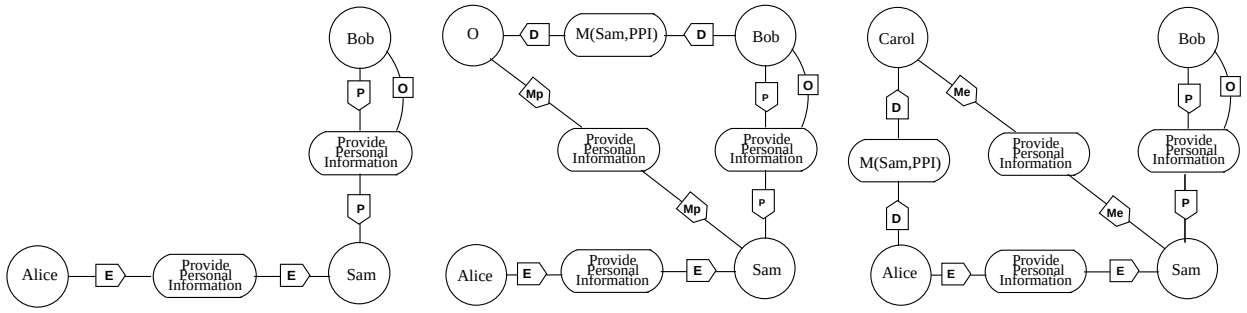
When services need to be delegated in the absence of trust, monitoring offers a surrogate for trust. According to Gans's et al. [11], the existence of distrust can be tolerated with an additional overhead of monitoring the untrustworthy delegatee. Here we refine Gans's et al. intuition and integrate it within our framework.

The goal of an actor playing the role of *monitor* is to check for the violation of trust[2]. The act of monitoring can be done by the delegator himself, or it can be delegated to another actor. Monitors can also be distinguished into **at-most** and **at-least** ones. Consider the situation presented in Fig. 1(a).

Example 6 *Suppose that there is no trust between Bob and Sam for the goal "maintain privacy", but Bob must delegate permission nonetheless. In this case, he depends (***D***) on the ombudsman (O) for monitoring if Sam transgresses his permissions. This is shown in Fig. 1(b)) with an at-most monitor (monitor for permission –* **Mp***) relationship between the ombudsman and Sam.*

Example 7 *If Alice is not confident that Sam will provide updated information, she may delegate to her secretary Carol the task of checking up on him to make sure new information about students is entered into the system. This is shown in Fig. 1(c)) with an at-least monitor (monitor for execution –* **Me***) relationship between Carol and Sam.*

[2]Indeed, monitoring could also be used for the evaluation of the fulfillment of a service assigned to a trusted actor.



(a) Delegation (b) At-most Monitor (c) At-least Monitor

Figure 1. Delegation and Monitoring

Another important distinction that emerges when we use a monitor is related to the type of service (goal, task or resource) for which monitoring is required. Let us assume that the service in Fig. 1 is a task (i.e., a specific sequence of actions). Here, the Monitor has to check if Sam executes the actions associated with the task.

Example 8 *To achieve the goal delegated to him in Example 7, Sam issues a letter to the head of each student secretariat office, so that student marks are entered into the system within 30 days from the date of final exams.*

What happens if Sam delegates the task or some of its subtasks to other actors? A possible solution to this problem is to extend monitoring across paths of delegation to the actor where actual execution takes place. According to this solution, there will be a monitor relationship between the Monitor and all intermediate actors involved in the execution of at least some part of the task.

Example 9 *To reach the objective of entering marks within 30 days, secretariat offices require that professors submit marks within this period. This is a further step of delegation of execution. To monitor this delegation, the actor responsible at the office, may also assign the task of reminding professors that they must return mark lists on time.*

Notice that monitoring as such is not a primitive construct in our framework. Instead, it is pattern that can be realized in terms of other constructs. In all cases, however, every service will either be delegated during the design process to a trusted actor, or it will be delegated to an untrusted one, in which case the delegatee will be monitored by a trusted actor.

In the formal model, monitoring is formalized in terms of additional axioms that ensure that an actor is confident that a service will be executed or a permission will not be abused even if existing trust relations offer no support for this.

Once we see monitoring as a simple design solution (essentially, a security pattern) we can treat monitoring goals like other goals in that they can be refined and delegated. Trust relationships linked to monitoring can then be captured with existing constructs. For example, monitoring often requires having permission to access monitored data or personnel. This itself may create problems of permission and authorization that can be modeled in the framework.

6 Formalization

As done in [13, 14] as well as [7, 18, 23], we use Datalog [1] to formalize the new concepts we have introduced in order to automatically verify the correctness and consistency of functional and security requirements. A Datalog program is a set of rules of the form L:- $L_1 \wedge ... \wedge L_n$, where L (called the head of the rule) is a positive literal and $L_1, ..., L_n$ are literals (called the body of the rule). Intuitively, the rule states that if $L_1, ..., L_n$ are true then L must be true. In Datalog, negation is treated as negation as failure: if it is not possible to infer that an atom is true, it is inferred that it is false.

We first present the predicates used for the formalization. Table 1 extends the predicates already presented in [13, 14] introducing new ones for execution, permission and monitoring[3]. To make the new predicates as generic as possible, we use the first argument of each predicate as a type parameter. Thus, delegate, delegateChain, and monitoring can be of type $t \in \{exec, perm\}$; trust, trustChain can have types $t \in \{exec, perm, mon\}$; as well, confident can have types $t \in \{satisfy, exec, owner\}$. For the same reason, predicates take as arguments generic services (i.e., goals, tasks and resources)[4].

6.1 Formal Model for Execution

The predicate requests(a, s) holds if actor a wants service s fulfilled, while provides(a, s) holds if actor a has the capability to fulfill service s. The predicate

[3]Monitoring is treated as a defined predicate.

[4]For resources the subgoal relation needs to be replaced by a part-of relation.

General predicates
delegate(Type : t, Actor : a, Actor : b, Service : s)
delegateChain(Type : t, Actor : a, Actor : b, Service : s)
trust(Type : t, Actor : a, Actor : b, Service : s)
trustChain(Type : t, Actor : a, Actor : b, Service : s)
monitoring(Type : t, Actor : a, Actor : b, Service : s)
confident(Type : t, Actor : a, Service : s)
Specific for execution
requests(Actor : a, Service : s)
provides(Actor : a, Service : s)
should_do(Actor : a, Service : s)
can_satisfy(Actor : a, Service : s)
Specific for Permission
owns(Actor : a, Service : s)
has_per(Actor : a, Service : s)
Goal refinement
goal(Service : s)
subgoal(Service : s_1, Service : s_2)
OR_subgoal(Service : s_1, Service : s_2)
AND_subgoal(Service : s_1, Service : s_2)
AND_decomp(Service : s_1, Service : s_2, Service : s_3)

Table 1. Predicates

delegate$(exec, a, b, s)$ holds if actor a delegates[5] the execution of service s to actor b. Actor a is called *delegator*; actor b is called *delegatee*. The predicate trust$(exec, a, b, s)$ holds is actor a trusts that actor b at least fulfills service s. Actor a is called *trustor*; actor b is called *trustee*. The predicate trust(mon, a, b, s) holds if actor a trusts that actor b monitors whether service s will be satisfied. The predicate monitoring$(exec, a, b, s)$ holds if actor a monitors if actor b can satisfy at least service s.

Other predicates define properties that will be used during formal analysis. The predicates delegateChain$(exec, a, b, s)$ and trustChain$(exec, a, b, s)$ hold if there is a delegation and a trust chain respectively, between actor a and actor b. The predicate should_do(a, s) identify actors who should directly fulfill a service. The predicate can_satisfy(a, s) holds if actor a can satisfy service s. The predicate confident$(satisfy, a, s)$ holds if actor a is confident that service s can be satisfied. Finally, we have predicates for goal/task refinement and resource decomposition. Their semantics and axiomatization are straight-forward.

The axiomatization of predicates for execution is shown in Table 2. The first batch of axioms deals with delegation and trust: E1 and E2 build a delegation chain of execution; E3 and E4 (M1 and M2) build a trust chain for execution (monitoring); E5 builds chains over monitoring steps. E6 and M4 have chains propagate from a service to its parts. According to E6 trust of execution flows top-down with respect to service decomposition. Likewise, axiom M4 for monitoring states that trustChain flows top-down with respect to service decomposition. M5 states that if an actor who is monitored for a service further delegates the service to another, then the monitoring is transferred to the delegatee; that is, monitoring is transferred along with delegations to ensure that a service is fulfilled.

The remaining axioms define global properties of the model. E7 and E8 state that an actor has to execute the service if he provides a service and if either some actor delegates the service to him, or he himself aims for the service. E9 and E10 state that an actor who aims for a service, can fulfill the service if either he provides it or has delegated it to someone who can fulfill it. Service decompositions are accounted for through axioms E11 and E12. If an actor can satisfy at least one of the or-sub-goals/tasks of a goal/task, then he can satisfy the root goal/task. Dual axiom holds for and-decompositions.

The notion of confidence is captured by axioms E13-E16. Actor a, who aims at service s, is confident that s will be fulfilled if he knows that all delegations have been done to trusted or monitored actors and that the actors have necessary permission. Axioms E15 and E16 specify how confidence is propagated upwards along a service decomposition hierarchy.

6.2 Formal Model for Permission

In Table 1 we include predicates for modeling permission. The first set of predicates corresponds to the relations used by the requirements engineer. The predicate owns(a, s) holds if actor a owns service s: the owner of a service has full authority concerning access and usage of the services, and can delegate this authority to other actors. The intuition is that delegate$(perm, a, b, s)$ holds if actor a at most delegates to actor b permission to fulfill service s. The predicate trust$(perm, a, b, s)$ holds is actor a trusts that actor b with the permission to fulfill service s. The predicate monitoring$(perm, a, b, s)$ is the dual of its execution counterpart.

Other predicates are used to define interesting properties used in formal analysis. The predicates delegateChain$(perm, a, b, s)$ and trustChain$(perm, a, b, s)$ hold if there is a delegation, resp. a trust chain of permission among actor a and actor b. The basic idea of has_per sums up the possible ways in which an actor can secure permission on a service: either directly or by delegation. From the point of view of the owner, confidence means that the owner is confident that the permission that he has delegated will not be misused. Alternatively, the owner is confident that he has delegated permission only to trusted or monitored agents. This means that even if there is one

[5]For the sake of simplicity we do not deal with the question of depth here. See Li et al. [17] for an account of delegation with depth. What has emerged from several case studies is that depth is less important than qualifications such as "only to members of the same office".

Delegation	
E1	$\mathsf{delegateChain}(exec, A, B, S) \leftarrow \mathsf{delegate}(exec, A, B, S)$
E2	$\mathsf{delegateChain}(exec, A, C, S) \leftarrow \mathsf{delegate}(exec, A, B, S) \wedge \mathsf{delegateChain}(exec, B, C, S)$
Trust	
E3	$\mathsf{trustChain}(exec, A, B, S) \leftarrow \mathsf{trust}(exec, A, B, S)$
E4	$\mathsf{trustChain}(exec, A, C, S) \leftarrow \mathsf{trust}(exec, A, B, S) \wedge \mathsf{trustChain}(exec, B, C, S)$
E5	$\mathsf{trustChain}(exec, A, C, S) \leftarrow \mathsf{trustChain}(mon, A, B, S) \wedge \mathsf{monitoring}(exec, M, C, S)$
E6	$\mathsf{trustChain}(exec, A, B, S_1) \leftarrow \mathsf{subgoal}(S, S_1) \wedge \mathsf{trustChain}(exec, A, B, S)$
M1	$\mathsf{trustChain}(mon, A, B, S) \leftarrow \mathsf{trust}(mon, A, B, S)$
M2	$\mathsf{trustChain}(mon, A, C, S) \leftarrow \mathsf{trust}(mon, A, B, S) \wedge \mathsf{trustChain}(mon, B, C, S)$
M3	$\mathsf{trustChain}(mon, A, C, S) \leftarrow \mathsf{trustChain}(exec, A, B, S) \wedge \mathsf{trustChain}(mon, B, C, S)$
M4	$\mathsf{trustChain}(mon, A, B, S_1) \leftarrow \mathsf{subgoal}(S, S_1) \wedge \mathsf{trustChain}(mon, A, B, S)$
Monitoring	
M5	$\mathsf{monitoring}(exec, M, B, S_1) \leftarrow \mathsf{delegateChain}(exec, A, B, S_1) \wedge \mathsf{monitoring}(exec, M, A, S) \wedge \mathsf{subgoal}(S_1, S)$
Should do	
E7	$\mathsf{should\_do}(A, S) \leftarrow \mathsf{delegateChain}(exec, B, A, S) \wedge \mathsf{provides}(A, S)$
E8	$\mathsf{should\_do}(A, S) \leftarrow \mathsf{requests}(A, S) \wedge \mathsf{provides}(A, S)$
Can satisfy	
E9	$\mathsf{can\_satisfy}(A, S) \leftarrow \mathsf{should\_do}(A, S)$
E10	$\mathsf{can\_satisfy}(A, S) \leftarrow \mathsf{delegate}(exec, A, B, S) \wedge \mathsf{can\_satisfy}(B, S)$
E11	$\mathsf{can\_satisfy}(A, S) \leftarrow \mathsf{OR\_subgoal}(S_1, S) \wedge \mathsf{can\_satisfy}(A, S_1)$
E12	$\mathsf{can\_satisfy}(A, S) \leftarrow \mathsf{AND\_decomp}(S, S_1, S_2) \wedge \mathsf{can\_satisfy}(A, S_1) \wedge \mathsf{can\_satisfy}(A, S_2)$
Confident to can satisfy	
E13	$\mathsf{confident}(satisfy, A, S) \leftarrow \mathsf{should\_do}(A, S)$
E14	$\mathsf{confident}(satisfy, A, S) \leftarrow \mathsf{delegateChain}(exec, A, B, S) \wedge \mathsf{trustChain}(exec, A, B, S) \wedge \mathsf{confident}(satisfy, B, S)$
E15	$\mathsf{confident}(satisfy, A, S) \leftarrow \mathsf{OR\_subgoal}(S_1, S) \wedge \mathsf{confident}(satisfy, A, S_1)$
E16	$\mathsf{confident}(satisfy, A, S) \leftarrow \mathsf{AND\_decomp}(S, S_1, S_2) \wedge \mathsf{confident}(satisfy, A, S_1) \wedge \mathsf{confident}(satisfy, A, S_2)$

Table 2. Axioms for execution

untrusted or unmonitored delegation, then the owner could be uneasy about the likely misuse of his permissions. So, an owner is confident, if there is no likely misuse of his permission. It can be seen that there is an intrinsic double negation in the statement. So we try to model it using a predicate $\mathsf{diffident}(a, s)$. At any point of delegation of permission, the delegating agent is diffident, if the delegation is being done to an agent who is neither trusted not monitored or if the delegatee could be diffident himself. In this way, $\mathsf{confident}(owner, a, s)$ holds if owner a is confident to give the permission on service s only to trusted actors.

Table 3 presents the axioms for permission. P1 and P2 build a delegation chain of permission; P3 and P4 build a trust chain for permission; P5 builds trust chains over monitoring steps. P6 has the chain propagate through subgoals. If an actor trusts that another will not overstep the set of actions required to fulfill a part of a service, then the first can trust the last will not overstep the set of actions required to fulfill the service. Essentially, trust of permission flows bottom-up with respect to goal refinements. M6 is used to build a trust chain for monitor. M7 states that if an actor under monitoring delegates a service to another, then the monitor have to watch for the delegatee, that is, the monitor follows the delegation. The owner of a service has full authority concerning access and disposition of it. Thus, P7 states that if an actor owns a service, he has it. P8 states that the delegatee has the service. The notion of confidence and diffidence that we have sketched above is captured by the axioms P10-P13.

6.3 Combining Execution and Permission

The checking of complex properties requires reasoning with both execution and permission. In Table 4 we present the notions from both the point of view of the requester and the point of view of the owner. The predicate $\mathsf{can\_execute}(a, s)$ holds if actor a can see service s fulfilled. The predicate $\mathsf{confident}(exec, a, s)$ holds if actor a is confident that service s will be fulfilled. This is the case if actor a knows that all delegations have been done to trusted or monitored actors and that the actors who will ultimately execute the service, have permission to do so. This is done using the axioms Ax5-Ax6. Goal refinements are taken care of by using axioms Ax7-Ax8 in a straight-forward way.

Owners may wish to delegate permissions to providers only if the latter actually do need the permission. The last part of Table 4 defines the predicates that are necessary to analyze *need-to-know* properties. As a result of absence of diffidence, the owner can be confident that his permission

Delegation	
P1	$\mathsf{delegateChain}(perm, A, B, S) \leftarrow \mathsf{delegate}(perm, A, B, S)$
P2	$\mathsf{delegateChain}(perm, A, C, S) \leftarrow \mathsf{delegate}(perm, A, B, S) \wedge \mathsf{delegateChain}(perm, B, C, S)$
Trust	
P3	$\mathsf{trustChain}(perm, A, B, S) \leftarrow \mathsf{trust}(perm, A, B, S)$
P4	$\mathsf{trustChain}(perm, A, C, S) \leftarrow \mathsf{trust}(perm, A, B, S) \wedge \mathsf{trustChain}(perm, B, C, S)$
P5	$\mathsf{trustChain}(perm, A, C, S) \leftarrow \mathsf{trustChain}(mon, A, B, S) \wedge \mathsf{monitoring}(perm, B, C, S)$
P6	$\mathsf{trustChain}(perm, A, B, S) \leftarrow \mathsf{subgoal}(S, S_1) \wedge \mathsf{trustChain}(perm, A, B, S_1)$
M6	$\mathsf{trustChain}(mon, A, C, S) \leftarrow \mathsf{trustChain}(perm, A, B, S) \wedge \mathsf{trustChain}(mon, B, C, S)$
Monitoring	
M7	$\mathsf{monitoring}(perm, M, B, S_1) \leftarrow \mathsf{delegateChain}(perm, A, B, S_1) \wedge \mathsf{monitoring}(perm, M, A, S) \wedge \mathsf{subgoal}(S_1, S)$
Has permission	
P7	$\mathsf{has\_per}(A, S) \leftarrow \mathsf{owns}(A, S)$
P8	$\mathsf{has\_per}(A, S) \leftarrow \mathsf{delegateChain}(perm, B, A, S) \wedge \mathsf{has\_per}(B, S)$
P9	$\mathsf{has\_per}(A, S_1) \leftarrow \mathsf{subgoal}(S_1, S) \wedge \mathsf{has\_per}(A, S)$
Owner is confident to give the service to trusted actors	
P10	$\mathsf{confident}(owner, A, S) \leftarrow \mathsf{owns}(A, S) \wedge \mathsf{not}\ \mathsf{diffident}(A, S)$
P11	$\mathsf{diffident}(A, S) \leftarrow \mathsf{delegateChain}(perm, A, B, S) \wedge \mathsf{diffident}(B, S)$
P12	$\mathsf{diffident}(A, S) \leftarrow \mathsf{delegateChain}(perm, A, B, S) \wedge \mathsf{not}\ \mathsf{trustChain}(perm, A, B, S)$
P13	$\mathsf{diffident}(A, S) \leftarrow \mathsf{subgoal}(S_1, S) \wedge \mathsf{diffident}(A, S_1)$

Table 3. Axioms for permission

Can see the service fulfilled (can execute)	
Ax1	$\mathsf{can\_execute}(A, S) \leftarrow \mathsf{should\_do}(A, S) \wedge \mathsf{has\_per}(A, S)$
Ax2	$\mathsf{can\_execute}(A, S) \leftarrow \mathsf{delegateChain}(exec, A, B, S) \wedge \mathsf{can\_execute}(B, S)$
Ax3	$\mathsf{can\_execute}(A, S) \leftarrow \mathsf{OR\_subgoal}(S_1, S) \wedge \mathsf{can\_execute}(A, S_1)$
Ax4	$\mathsf{can\_execute}(A, S) \leftarrow \mathsf{AND\_decomp}(S, S_1, S_2) \wedge \mathsf{can\_execute}(A, S_1) \wedge\ \mathsf{can\_execute}(A, S_2)$
Confident to see the service fulfilled (confident to execute)	
Ax5	$\mathsf{confident}(exec, A, S) \leftarrow \mathsf{should\_do}(A, S) \wedge \mathsf{has\_per}(A, S)$
Ax6	$\mathsf{confident}(exec, A, S) \leftarrow \mathsf{delegateChain}(exec, A, B, S) \wedge \mathsf{trustChain}(exec, A, B, S) \wedge \mathsf{confident}(exec, B, S)$
Ax7	$\mathsf{confident}(exec, A, S) \leftarrow \mathsf{OR\_subgoal}(S_1, S) \wedge \mathsf{confident}(exec, A, S_1)$
Ax8	$\mathsf{confident}(exec, A, S) \leftarrow \mathsf{AND\_decomp}(S, S_1, S_2) \wedge \mathsf{confident}(exec, A, S_1) \wedge\ \mathsf{confident}(exec, A, S_2)$
Need to know	
Ax9	$\mathsf{need\_to\_have\_perm}(A, S) \leftarrow \mathsf{should\_do}(A, S)$
Ax10	$\mathsf{need\_to\_have\_perm}(A, S) \leftarrow \mathsf{delegate}(perm, A, B, S) \wedge \mathsf{not}\ \mathsf{other\_delegater}(A, B, S) \wedge \mathsf{need\_to\_have\_perm}(B, S)$
Ax11	$\mathsf{other\_delegater}(A, B, S) \leftarrow \mathsf{delegate}(perm, C, B, S) \wedge \mathsf{need\_to\_have\_perm}(C, S) \wedge A \neq C$

Table 4. Axioms Involving both permission and execution

will not be misused. But has this permission reached the actors who actually *need* it? The owner might also want to ensure that there has been no unwanted delegation of permission. This can be achieved by identifying the actors who actually *need-to-know* (or rather *need-to-have*) the permission. This set of axioms captures also the possibility of having alternate paths of permission delegations. In this case the formal analysis will not yield one model but multiple models in which only one path of delegation is labeled by the need-to-have property and the others are not.

Example 10 (Figure 2) *Alice and Carol (7 and 8) have both received consent (permission) from Bob (1) for using his personal data. In turn, they both delegate it to the faculty secretariat (3), which must have the permission to provide the data to Paul (6), the university tutor who is responsible for providing personal counseling to Bob. In this case only one of either Alice or Carol needs to have the permission.*

7 Analysis and Verification

We use the DLV system[6] to verify security properties with respect to a Secure Tropos model. In Table 5 we use the $A \Rightarrow? B$ to mean that one must check that each time A holds it is desirable that B also holds. In Datalog this can be represented as the constraint :- A, not B. If the set of features is not consistent, i.e., they cannot all be simultaneously satisfied, the system is inconsistent, and hence it is not secure. This technique also allows us to check that our proposed axioms are indeed consistent.

[6] http://www.dbai.tuwien.ac.at/proj/dlv

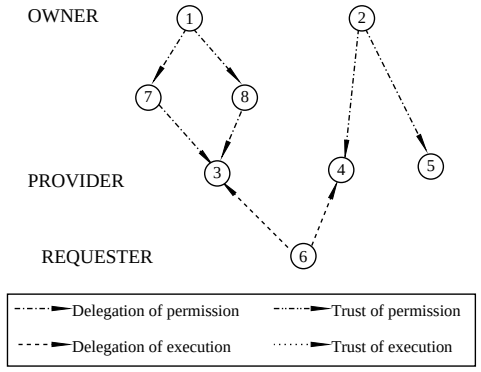
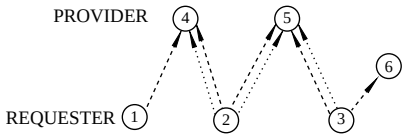


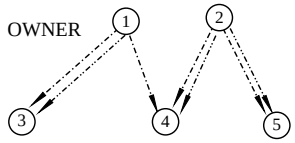
Figure 2. Need-to-Know and Multiple Permissions Paths

Execution	
Pro1	$\mathsf{delegateChain}(exec, A, B, S)$ $\Rightarrow?\ \mathsf{trustChain}(exec, A, B, S)$
Pro2	$\mathsf{requests}(A, S) \Rightarrow?\mathsf{can\_satisfy}(A, S)$
Pro3	$\mathsf{requests}(A, S) \Rightarrow?\mathsf{confident}(satisfy, A, S)$
Pro4	$\mathsf{should\_do}(A, S) \Rightarrow?\mathsf{not\ delegateChain}(exec, A, B, S)$
Permission	
Pro5	$\mathsf{delegateChain}(perm, A, B, S)$ $\Rightarrow?\ \mathsf{trustChain}(perm, A, B, S)$
Pro6	$\mathsf{owns}(A, S) \Rightarrow?\ \mathsf{confident}(owner, A, S)$
Pro7	$\mathsf{owns}(A, S) \Rightarrow?$ $\mathsf{not\ delegateChain}(perm, B, A, S) \wedge A \neq B$
Execution & Permission	
Pro8	$\mathsf{requests}(A, S) \Rightarrow?\mathsf{can\_execute}(A, S)$
Pro9	$\mathsf{requests}(A, S) \Rightarrow?\mathsf{confident}(exec, A, S)$
Pro10	$\mathsf{owns}(A, S) \Rightarrow?\mathsf{need\_to\_have\_perm}(A, S)$
Pro11	$\mathsf{owns}(A, S) \Rightarrow?$ $\mathsf{need\_to\_have\_perm}(A, S) \wedge \mathsf{confident}(owner, A, S)$

Table 5. Desirable Properties of a Design



(a) From requesters' viewpoint



(b) From owners' viewpoint

Figure 3. Design for execution and permission

Pro1 states that if there is a delegation chain either the delegater trusts the delegatee or there is a monitor and the delegator trust the monitor. Pro2 states that a requester can satisfy his goals, and Pro3 states that a requester wants to be confident to satisfy the service.

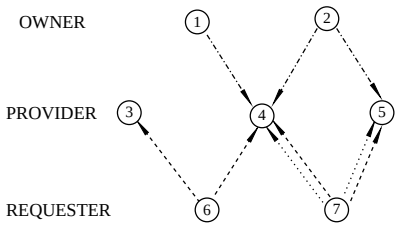


Figure 4. Owner vs Requester

Example 11 (Figure 3(a)) *Bob and Bert (1 and 2) need counseling. They can receive it (formal relation can satisfy) because they delegate the execution to Paul and Peter (4 and 5), while Bill (3) cannot receive all the advice he needs because he requested it from Alice (6), who is not able to provide counseling on faculty matters.*

Bob is also confident that he will receive all counseling he needs since he delegates the execution to Paul and Peter (4 and 5) whom he trusts, while Bert is not confident since he delegates to Paul (4) whom he does not trust.

Pro4 states that if an actor provides a service and if either some actor delegates this service to him or he himself requests the service, then he has to execute the service without further delegation. Pro5 states that if there is a delegation chain, either the delegator trusts the delegatee or there is a monitor. Pro6 states that the owner of the service has to be confident to give the service to trusted actors, and Pro7 states that a service cannot be delegated back to its owner.

Example 12 (Figure 3(b)) *Bob and Bert (1 and 2) need to provide their personal data in order to get effective counseling. Bob is confident about his personal data since he delegates permissions to Paul and Peter (4 and 5), whom he trusts to only use the data for counseling. On the other hand, Bert is not confident on his data since he delegates it to Paul (4) whom he does not trust to keep the information confidential.*

This is similar to the previous example on misplaced delegation (Example 11). The difference between the two lies in what is possible for poor Bert. In the former case he is afraid to receive bad advice (delegation of execution), in the latter his personal information may be misused.

The last part of Table 5 shows properties to verify at-most model and at-least model at the same time. Pro8 states that the requester has to can see the service fulfilled. Pro9 states that the requester has to be confident to see the service fulfilled.

Example 13 (Figure 4) *Bob and Bert (1 and 2) delegate permission to Sam (4). Moreover, Bob delegates permission to his parents (5). Paul (7), needs to get student information to provide accurate counseling, but he cannot directly ask them. Thus, he delegates the task of getting permission for the data to Sam. Paul could also ask permission for Bob's data from Bob's parents. So Paul can suppose that someone provides the personal information of his students. On the other hand, Peter (6), delegates execution of the task to provide his students' personal information to Carol (3), but the latter does not have permission to manage it. Thus, Carol cannot forward the information to Peter. Further, Paul delegates execution of the task to make available personal information of his students to the student information system. If he does not trust the system for this goal, then he is not confident he will get the personal information.*

8 Related Work and Conclusions

The work by Liu et al. [19] uses the goal-oriented i*/Tropos RE methodology to introduce goals such as "Security" or "Privacy", and proposes dependency analysis to check if the system is secure. In [3], general taxonomies for privacy are proposed for a standard goal oriented analysis. Another early RE example is [25], which presents a requirements process model, based upon reuse and templates, for security policies in a organization.

On the side of approaches explicitly intended for security, Jürjens has proposed UMLsec [16], an extension of low-level security mechanisms in UML, and the CORAS methodology for modeling risk and vulnerability [9]. In the same spirit, Lodderstedt et al. [20] propose an UML-based modeling language (SecureUML). Their approach is focused on modeling access control policies and integrating them into a model-driven software development process. One of the major limitations of all these proposals is that they treat security in system-oriented terms, and do not support the modeling and analysis of security requirements at an organizational level. In other words, they are targeted to model a computer system and the policies and access control mechanisms it supports. In contrast, to understand the problem of security engineering we need to model the organization and social relationships between all actors involved in the system.

Other approaches [22, 24, 26] propose to model the behavior of attachers. McDermott and Fox adapt use cases [22] to capture and analyze security requirements, and they call the adaption an abuse case model. An abuse case is an interaction between a system and one or more actors, where the results of the interaction are harmful to the system, or one of the stakeholders of the system. Guttorm and Opdahl [24] define misuse cases, the inverse of UML use cases, which describe functions that the system should not allow. Moving towards early requirements, the role of abuse-cases is played by Anti-Goals proposed by van Lamsweerde et al. [26].

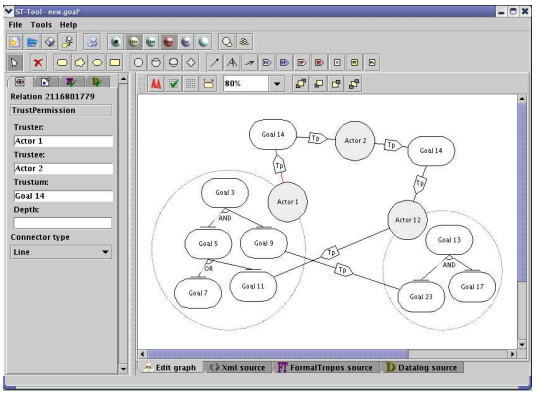


Figure 5. The ST-Tool

In this paper we have extended our previous work [13, 14] providing a comprehensive modeling framework for security requirements. In particular, the framework offers

- the notions of delegation and trust of execution and delegation and trust of permission, respectively in the form of at-least and at-most delegation and trust;
- the use of monitoring as design solution (pattern) to overcome the problem of lack of trust between actors;
- a comprehensive semantic model based on Datalog to ease translations from requirements into security policies and trust management systems using the same semantics (as already stated in [13]).

Our framework with all new features presented in this paper is supported by the ST-Tool[7] (Figure 5). ST-Tool is a graphical tool (implemented in Java) to support the design of (Secure) Tropos models. The tool allows system designers to draw (Secure) Tropos diagrams by selecting from the menu the desired (Secure) Tropos elements and to verify the correctness of the specification of the corresponding element. It also supports the automatic transformation of Secure Tropos graphical models into formal specifications, specifically both Datalog specification and Formal Tropos specification [10]. For every Tropos element it is possible to specify properties (i.e. creation-properties, invar-properties, fulfill-properties) with respect to the syntax of Formal Tropos, and the resulting specification is automatically displayed. A Datalog specification can also be generated and displayed along similar lines. In addition, the tool provides a user-friendly interface to the DLV system and permits a designer to select properties of each model and to specify additional security policies. The resulting Datalog specifications are automatically verified by the DLV system.

[7] Available on the web at `http://sesa.dit.unitn.it/sttool/`.

Future work plans include refining the proposed framework to the point where we can derive security services and mechanisms comparable to the standards ISO-7498-2 and ISO-17799.

9 Acknowledgments

This work has been partially funded by the IST programme of the EU Commission, through an FET under the IST-2001-37004 WASP project, by the FIRB programme of MIUR under the RBNE0195K5 ASTRO project, also by the MOSTRO and SMTPPs projects funded by the Provincial Authority of Trentino.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Computer Publishing, 2001.

[3] A. I. Antòn, J. B. Earp, and A. Reese. Analyzing Website privacy requirements using a privacy goal taxonomy. In *Proc. of RE'02*, pages 23– 31. IEEE Press, 2002.

[4] T. Aura. On the Structure of Delegation Networks. In *Proc. of 1998 CSFW*, pages 14–26. IEEE Press, 1998.

[5] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An Agent-Oriented Software Development Methodology. *JAAMAS*, 8(3):203–236, 2004.

[6] R. Crook, D. Ince, L. Lin, and B. Nuseibeh. Security Requirements Engineering: When Anti-requirements Hit the Fan. In *Proc. of RE'02*, pages 203–205. IEEE Press, 2002.

[7] J. DeTreville. Binder, a logic-based security language. In *Proc. of 2002 IEEE Symp. on Sec. and Privacy*, pages 95–103. IEEE Press, 2002.

[8] P. T. Devanbu and S. G. Stubblebine. Software engineering for security: a roadmap. In *Proc. of ICSE'00*, pages 227–239, 2000.

[9] R. Fredriksen, M. Kristiansenand, B. A. G. K. Stølen, T. A. Opperud, and T. Dimitrakos. The CORAS framework for a model-based risk management process. In *Proc. of SAFECOMP'02*, *LNCS 2434*, pages 94–105, 2002.

[10] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in tropos. In *Proc. of RE'01*, pages 174–181. IEEE Press, 2001.

[11] G. Gans, M. Jarke, S. Kethers, and G. Lakemeyer. Modeling the Impact of Trust and Distrust in Agent Networks. In *Proc. of AOIS'01*, pages 45–58, 2001.

[12] P. Giorgini, F. Massacci, and J. Mylopoulos. Requirement Engineering meets Security: A Case Study on Modelling Secure Electronic Transactions by VISA and Mastercard. In *Proc. of ER'03*, *LNCS 2813*, pages 263–276. Springer-Verlag, 2003.

[13] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Filling the gap between Requirements Engineering and Public Key/Trust Management Infrastructures. In *Proc. of EuroPKI'04*, *LNCS 3093*, pages 98–111. Springer-Verlag, 2004.

[14] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Requirements Engineering meets Trust Management: Model, Methodology, and Reasoning. In *Proc. of iTrust'04*, *LNCS 2995*, pages 176–190. Springer-Verlag, 2004.

[15] A. J. I. Jones and M. J. Sergot. A Formal Characterisation of Institutionalised Power. *J. of the Interest Group in Pure and Appl. Log.*, 4(3):429–445, 1996.

[16] J. Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2004.

[17] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *TISSEC*, 6(1):128–171, 2003.

[18] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of A Role-based Trust-management Framework. In *Proc. of 2002 IEEE Symp. on Sec. and Privacy*, pages 114–130. IEEE Press, 2002.

[19] L. Liu, E. S. K. Yu, and J. Mylopoulos. Security and Privacy Requirements Analysis within a Social Setting. In *Proc. of RE'03*, pages 151–161. IEEE Press, 2003.

[20] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proc. of UML'02*, *LNCS 2460*, pages 426–441. Springer-Verlag, 2002.

[21] F. Massacci, M. Prest, and N. Zannone. Using a Security Requirements Engineering Methodology in Practice: The compliance with the Italian Data Protection Legislation. *Comp. Standards & Interfaces*, 27(5):445–455, 2005. An extended version is available as Technical report DIT-04-103 at `eprints.biblio.unitn.it`.

[22] J. McDermott and C. Fox. Using Abuse Case Models for Security Requirements Analysis. In *Proc. of ACSAC'99*, pages 55–66. IEEE Press, 1999.

[23] P. Samarati and S. D. C. di Vimercati. Access Control: Policies, Models, and Mechanisms. In *FOSAD II*, *LNCS 2946*, pages 137–196. Springer-Verlag, 2001.

[24] G. Sindre and A. L. Opdahl. Eliciting Security Requirements by Misuse Cases. In *Proc. of TOOLS Pacific 2000*, pages 120–131. IEEE Press, 2000.

[25] A. Toval, A. Olmos, and M. Piattini. Legal requirements reuse: a critical success factor for requirements quality and personal data protection. In *Proc. of RE'02*, pages 95 –103. IEEE Press, 2002.

[26] A. van Lamsweerde, S. Brohez, R. De Landtsheer, and D. Janssens. From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering. In *Proc. of RHAS'03*, pages 49–56, 2003.

[27] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2001.